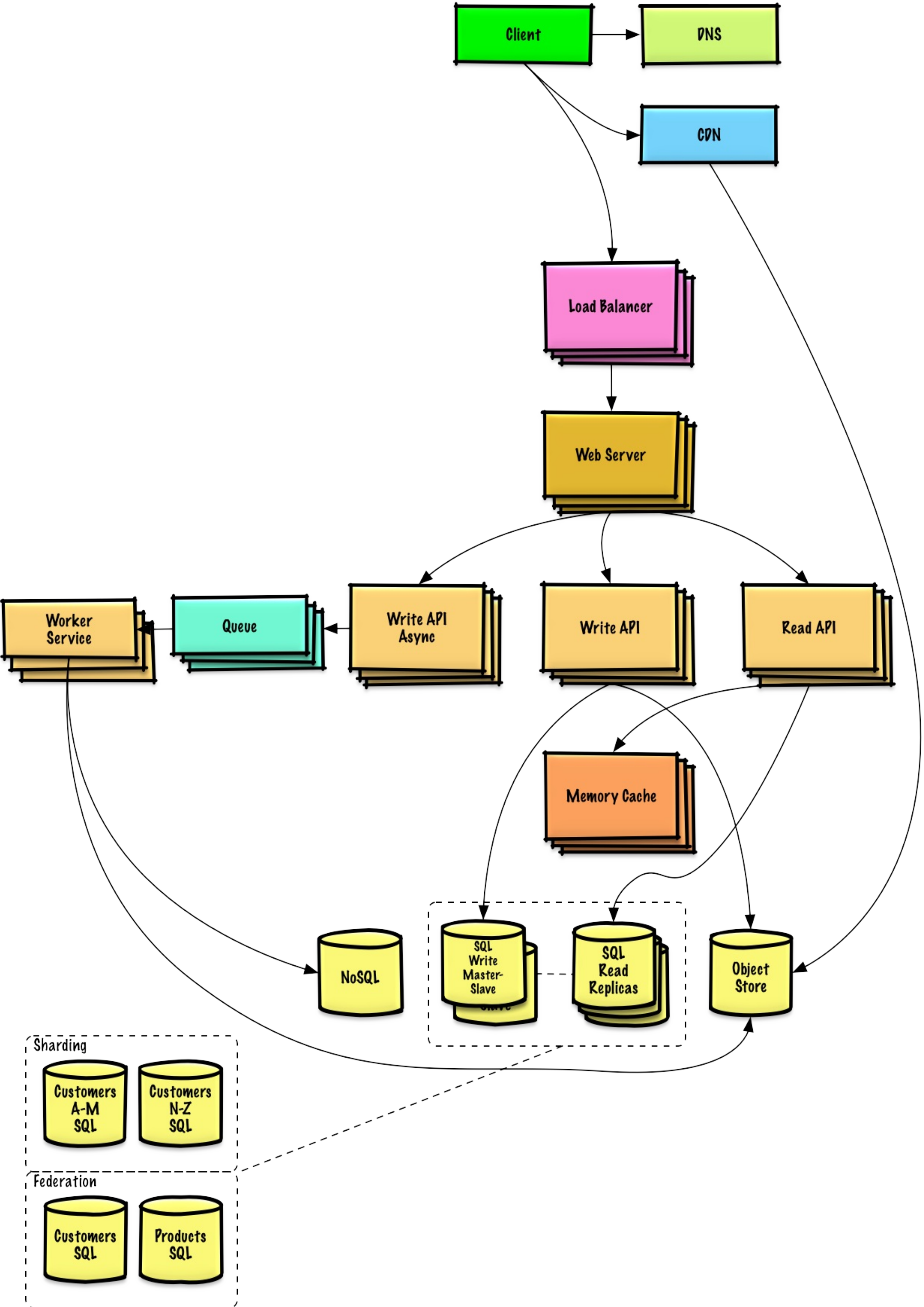


Client
DNS
CDN
Load Balancer
Web Server
Write API Async
Write API
Read API
Worker Service
Queue
Memory Cache
NoSQL
SQL Write Master-Slave
SQL Read Replicas
Object Store
Sharding
Customers A-M SQL
Customers N-Z SQL
Federation
Customers SQL
Products SQL

目錄

系统设计入门 1.1
目的 1.2
抽认卡 1.3
贡献 1.4
系统设计主题的索引 1.5
学习指引 1.6
如何处理一个系统设计的面试题 1.7
- 设计 Pastebin.com 1.7.1
- 设计 Twitter 时间线和搜索 1.7.2
- 设计一个网页爬虫 1.7.3
- 设计 Mint.com 1.7.4
- 为一个社交网络设计数据结构 1.7.5
- 为搜索引擎设计一个 key-value 储存 1.7.6
- 通过分类特性设计 Amazon 的销售排名 1.7.7
- 在 AWS 上设计一个百万用户级别的系统 1.7.8

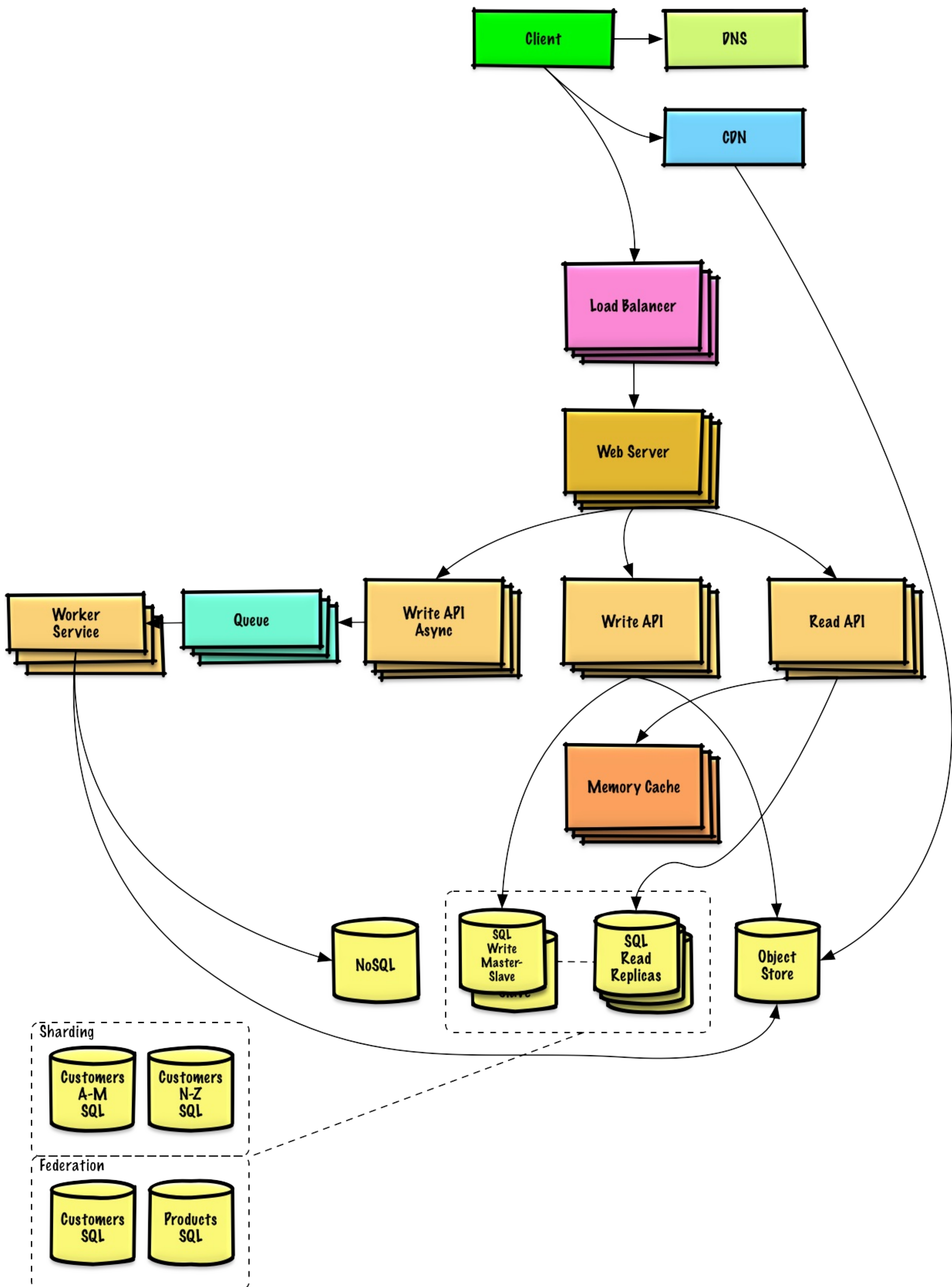
系统设计的面试题和解答 1.8
面向对象设计的面试问题及解答 1.9
系统设计主题：从这里开始 1.10
性能与可扩展性 1.11
延迟与吞吐量 1.12
可用性与一致性 1.13
一致性模式 1.14
可用性模式 1.15
域名系统 1.16
内容分发网络（CDN） 1.17
负载均衡器 1.18
反向代理（web 服务器） 1.19
应用层 1.20
数据库 1.21
缓存 1.22

异步 1.23
通讯 1.24
安全 1.25
附录 1.26
正在完善中 1.27
致谢 1.28
联系方式 1.29
许可 1.30

系统设计入门

> - 原文地址：github.com/donnemartin/system-design-primer
> - 译文出自：掘金翻译计划
> - 译者：XatMassacrE、L9m、Airmacho、xiaoyusilen、jifaxu、根号三
> - 这个 链接 用来查看本翻译与英文版是否有差别（如果你没有看到 README.md 发生变化，那就意味着这份翻译文档是最新的）。

English · 简体中文 **| Brazilian Portuguese · Turkish | Add Translation**


Client
DNS
CDN
Load Balancer
Web Server
Worker Service
Queue
Write API Async
Write API
Read API
Memory Cache
NoSQL
SQL Write Master-Slave
SQL Read Replicas
Object Store
Sharding
Customers A-M SQL
Customers N-Z SQL
Federation
Customers SQL
Products SQL

目的

> 学习如何设计大型系统。
>
> 为系统设计的面试做准备。

学习如何设计大型系统

学习如何设计可扩展的系统将会有助于你成为一个更好的工程师。

系统设计是一个很宽泛的话题。在互联网上，关于系统设计原则的资源也是多如牛毛。

这个仓库就是这些资源的组织收集，它可以帮助你学习如何构建可扩展的系统。

从开源社区学习

这是一个不断更新的开源项目的初期的版本。

欢迎贡献！

为系统设计的面试做准备

在很多科技公司中，除了代码面试，系统设计也是技术面试过程中的一个必要环节。

实践常见的系统设计面试题并且把你的答案和例子的解答进行对照：讨论，代码和图表。

面试准备的其他主题：

- 学习指引
- 如何处理一个系统设计的面试题
- 系统设计的面试题，含解答
- 面向对象设计的面试题，含解答
- 其它的系统设计面试题

抽认卡

这里提供的抽认卡堆使用间隔重复的方法，帮助你记忆关键的系统设计概念。

- 系统设计的卡堆
- 系统设计的练习卡堆
- 面向对象设计的练习卡堆

随时随地都可使用。

代码资源：互动式编程挑战

你正在寻找资源以准备编程面试吗？

2 0 85 00:16

Algorithm

A heap is a complete binary tree where each node is smaller than its children.

extract_min

```
          _5_
        /     \
       20      15
      / \     /  \
    22   40  25

Save the root as the value to be returned: 5
Move the right most element to the root: 25

          _25_
        /      \
       20       15
      / \      /  \
    22   40

Bubble down 25: Swap 25 and 15 (the smaller
child)

          _15_
        /      \
       20       25
      / \      /  \
    22   40
```

2h AGAIN | 9d HARD | 14d GOOD | 14d EASY

Coding
0 minutes left

2 0 91 00:15

```python
class Node(object):

    def __init__(self, data, next=None):
        self.next = next
        self.data = data

    def __str__(self):
        return self.data


class LinkedList(object):

    def __init__(self, head=None):
        self.head = head

    def __len__(self):
        curr = self.head
        counter = 0
        while curr is not None:
            counter += 1
            curr = curr.next
        return counter

    def insert_to_front(self, data):
        if data is None:
            return None
        node = Node(data, self.head)
        self.head = node
        return node

    def append(self, data):
        if data is None:
            return None
```

2h AGAIN | 1d GOOD

请查看我们的姐妹仓库互动式编程挑战，其中包含了一个额外的抽认卡堆：

- 代码卡堆

贡献

> 从社区中学习。

欢迎提交 PR 提供帮助：

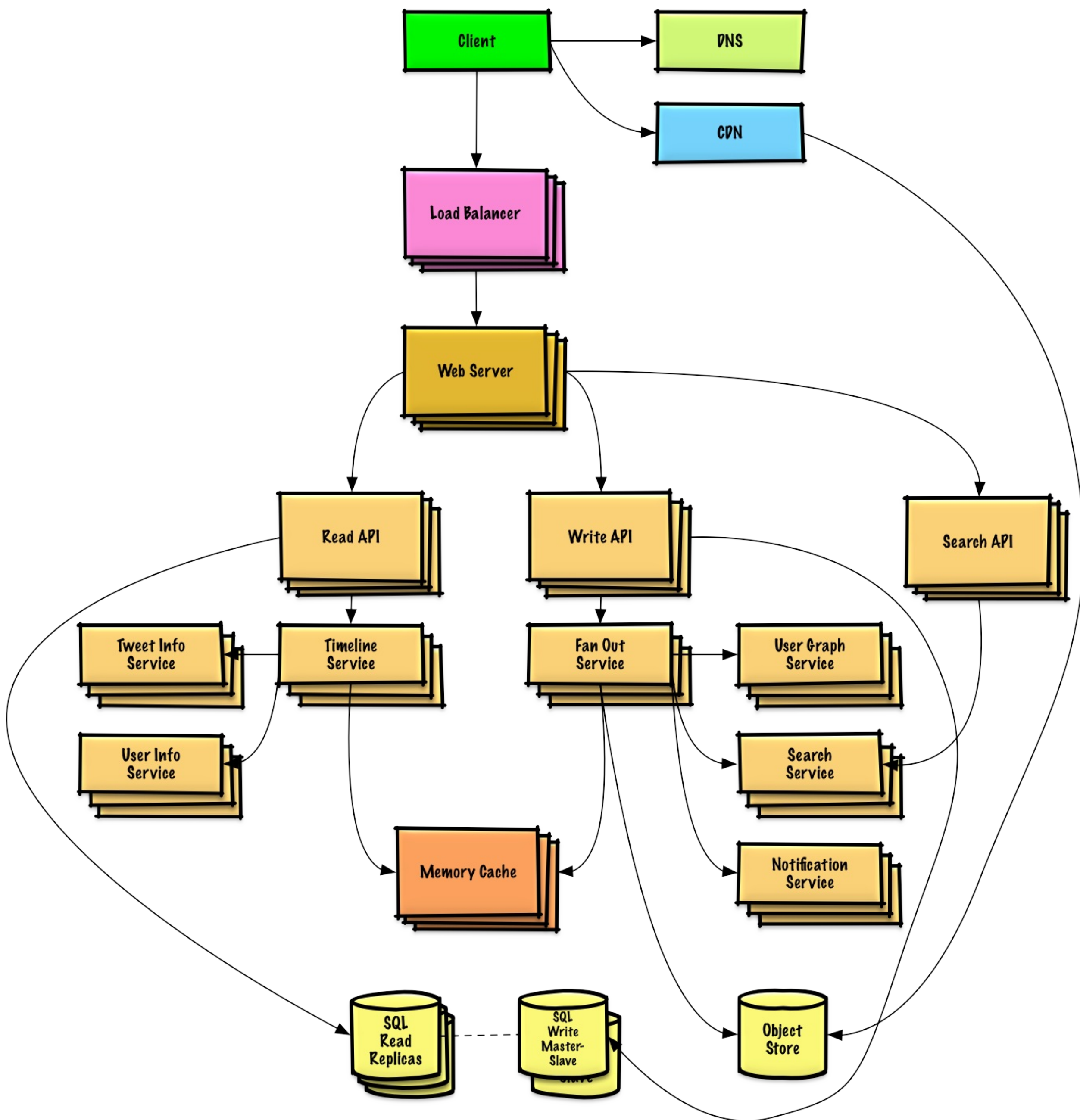
- 修复错误
- 完善章节
- 添加章节
- 帮助翻译

一些还需要完善的内容放在了正在完善中。

请查看贡献指南。

系统设计主题的索引

> 各种系统设计主题的摘要，包括优点和缺点。每一个主题都面临着取舍和权衡。
>
> 每个章节都包含着更的资源的链接。



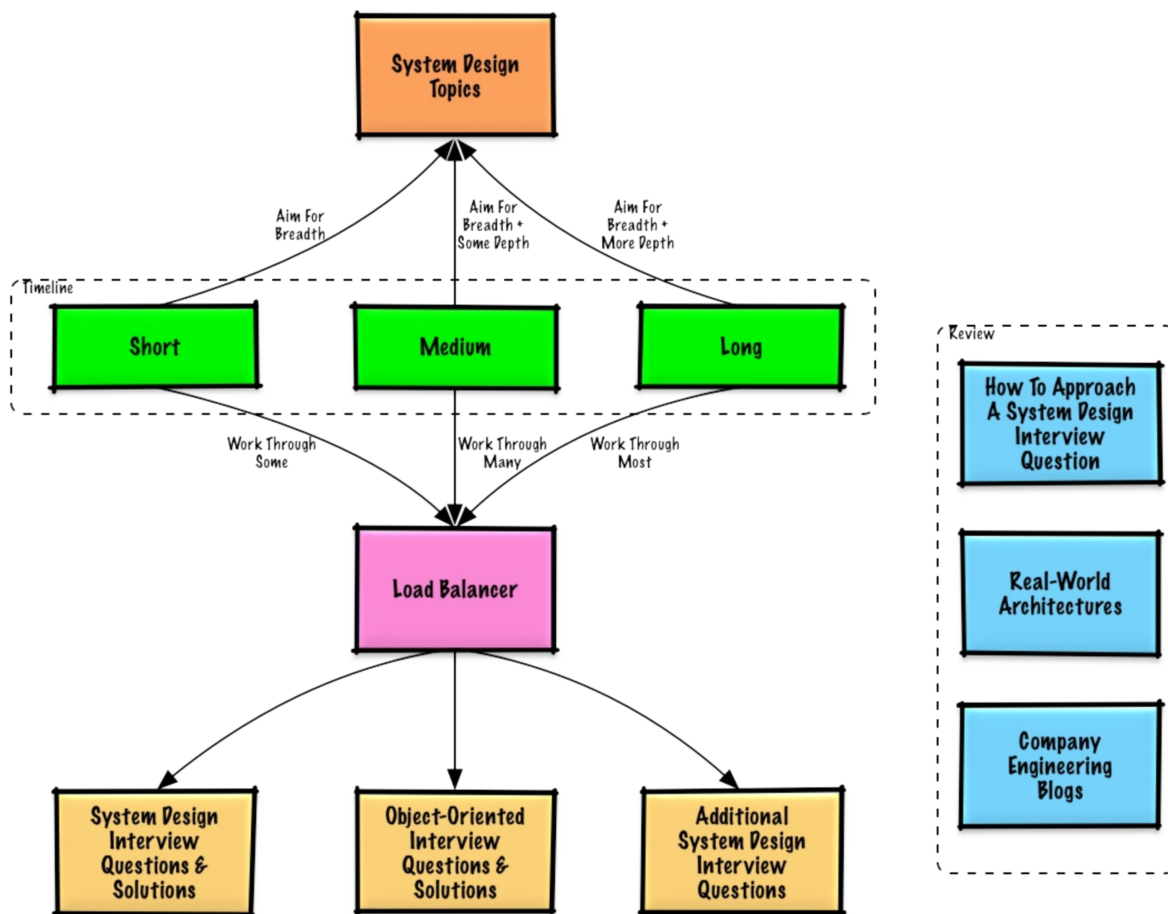
- 系统设计主题：从这里开始
    - 第一步：回顾可扩展性的视频讲座
    - 第二步： 回顾可扩展性的文章
    - 接下来的步骤
- 性能与拓展性
- 延迟与吞吐量

- 可用性与一致性
  - CAP 理论
    - CP - 一致性和分区容错性
    - AP - 可用性和分区容错性
- 一致模式
  - 弱一致性
  - 最终一致性
  - 强一致性
- 可用模式
  - 故障切换
  - 复制
- 域名系统
- CDN
  - CDN 推送
  - CDN 拉取
- 负载均衡器
  - 工作到备用切换（Active-passive）
  - 双工作切换（Active-active）
  - 四层负载均衡
  - 七层负载均衡
  - 水平扩展
- 反向代理（web 服务器）
  - 负载均衡与反向代理
- 应用层
  - 微服务
  - 服务发现
- 数据库
  - 关系型数据库管理系统（RDBMS）
    - Master-slave 复制集
    - Master-master 复制集
    - 联合
    - 分片
    - 非规范化
    - SQL 调优
  - NoSQL
    - Key-value 存储
    - 文档存储
    - 宽列存储
    - 图数据库
  - SQL 还是 NoSQL

- 缓存
  - 客户端缓存
  - CDN 缓存
  - Web 服务器缓存
  - 数据库缓存
  - 应用缓存
  - 数据库查询级别的缓存
  - 对象级别的缓存
  - 何时更新缓存
    - 缓存模式
    - 直写模式
    - 回写模式
    - 刷新
- 异步
  - 消息队列
  - 任务队列
  - 背压机制
- 通讯
  - 传输控制协议（TCP）
  - 用户数据报协议（UDP）
  - 远程控制调用协议（RPC）
  - 表述性状态转移（REST）
- 安全
- 附录
  - 2 的次方表
  - 每个程序员都应该知道的延迟数
  - 其它的系统设计面试题
  - 真实架构
  - 公司的系统架构
  - 公司工程博客
- 正在完善中
- 致谢
- 联系方式
- 许可

学习指引

> 基于你面试的时间线（短、中、长）去复习那些推荐的主题。



问：对于面试来说，我需要知道这里的所有知识点吗？

答：不，如果只是为了准备面试的话，你并不需要知道所有的知识点。

在一场面试中你会被问到什么取决于下面这些因素：

- 你的经验
- 你的技术背景
- 你面试的职位
- 你面试的公司
- 运气

那些有经验的候选人通常会被期望了解更多的系统设计的知识。架构师或者团队负责人则会被期望了解更多除了个人贡献之外的知识。顶级的科技公司通常也会有一次或者更多的系统设计面试。

面试会很宽泛的展开并在几个领域深入。这回帮助你了解一些关于系统设计的不同的主题。基于你的时间线，经验，面试的职位和面试的公司对下面的指导做出适当的调整。

- 短期 - 以系统设计主题的广度为目标。通过解决一些面试题来练习。
- 中期 - 以系统设计主题的广度和初级深度为目标。通过解决很多面试题来练习。
- 长期 - 以系统设计主题的广度和高级深度为目标。通过解决大部分面试题来联系。

	短期	中期	长期
阅读 系统设计主题 以获得一个关于系统如何工作的宽泛的认识	:+1:	:+1:	:+1:
阅读一些你要面试的公司工程博客的文章	:+1:	:+1:	:+1:
阅读 真实架构	:+1:	:+1:	:+1:
复习 如何处理一个系统设计面试题	:+1:	:+1:	:+1:
完成 系统设计的面试题和解答	一些	很多	大部分
完成 面向对象设计的面试题和解答	一些	很多	大部分
复习 其它的系统设计面试题	一些	很多	大部分

如何处理一个系统设计的面试题

系统设计面试是一个开放式的对话。他们期望你去主导这个对话。

你可以使用下面的步骤来指引讨论。为了巩固这个过程，请使用下面的步骤完成系统设计的面试题和解答这个章节。

第一步：描述使用场景，约束和假设

把所有需要的东西聚集在一起，审视问题。不停的提问，以至于我们可以明确使用场景和约束。讨论假设。

- 谁会使用它？
- 他们会怎样使用它？
- 有多少用户？
- 系统的作用是什么？
- 系统的输入输出分别是什么？
- 我们希望处理多少数据？
- 我们希望每秒钟处理多少请求？
- 我们希望的读写比率？

第二步：创建概要设计

使用所有重要的组件来描绘出一个概要设计。

- 画出主要的组件和连接
- 证明你的想法

第三步：设计核心组件

对每一个核心组件进行详细深入的分析。举例来说，如果你被问到设计一个 url 缩写服务，开始讨论：

- 生成并储存一个完整 url 的 hash
    - MD5 和 Base62
    - Hash 碰撞
    - SQL 还是 NoSQL
    - 数据库模型
- 将一个 hashed url 翻译成完整的 url
    - 数据库查找
- API 和面向对象设计

第四步：度量设计

确认和处理瓶颈以及一些限制。举例来说就是你需要下面的这些来完成拓展性的议题吗？

- 负载均衡
- 水平拓展
- 缓存
- 数据库分片

论述可能的解决办法和代价。每件事情需要取舍。可以使用可拓展系统的设计原则来处理瓶颈。

信封背面的计算

你或许会被要求通过手算进行一些估算。涉及到的附录涉及到的是下面的这些资源：

- 使用信封的背面做计算
- 2 的次方表
- 每个程序员都应该知道的延迟数

相关资源和延伸阅读

查看下面的链接以获得我们期望的更好的想法：

- 怎样通过一个系统设计的面试
- 系统设计的面试
- 系统架构与设计的面试简介

设计 Pastebin.com（或 Bit.ly）

注意：这个文档中的链接会直接指向系统设计主题索引中的有关部分，以避免重复的内容。你可以参考链接的相关内容，来了解其总的要点、方案的权衡取舍以及可选的替代方案。

除了粘贴板需要存储的是完整的内容而不是短链接之外，设计 **Bit.ly**是与本文类似的一个问题。

第一步：简述用例与约束条件

> 搜集需求与问题的范围。 提出问题来明确用例与约束条件。 讨论假设。

我们将在没有面试官明确说明问题的情况下，自己定义一些用例以及限制条件。

用例

我们将把问题限定在仅处理以下用例的范围中

- 用户输入一些文本，然后得到一个随机生成的链接
  - 过期时间
    - 默认为永不过期
    - 可选设置为一定时间过期
- 用户输入粘贴板中的 url，查看内容
- 用户是匿名访问的
- 服务需要能够对页面进行跟踪分析
  - 月访问量统计
- 服务将过期的内容删除
- 服务有着高可用性

不在用例范围内的有

- 用户注册了账号
  - 用户通过了邮箱验证
- 用户登录已注册的账号
  - 用户编辑他们的文档
- 用户能设置他们的内容是否可见
- 用户是否能自行设置短链接

限制条件与假设

提出假设

- 网络流量不是均匀分布的
- 生成短链接的速度必须要快
- 只允许粘贴文本
- 不需要对页面预览做实时分析
- 1000 万用户
- 每个月 1000 万次粘贴
- 每个月 1 亿次读取请求
- 10:1 的读写比例

计算用量

如果你需要进行粗略的用量计算，请向你的面试官说明。

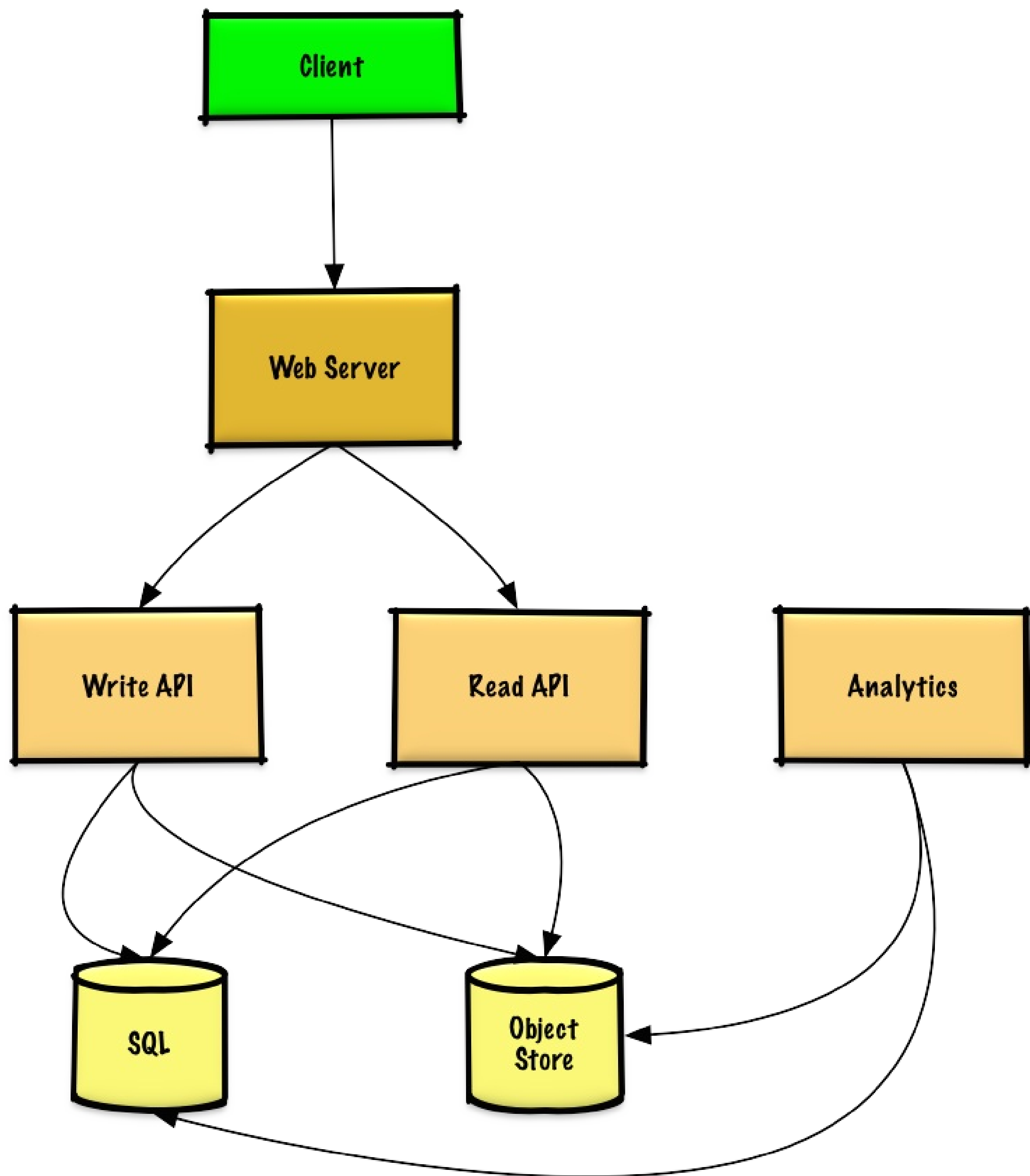
- 每次粘贴的用量
    - 1 KB 的内容
    - `shortlink` - 7 字节
    - `expiration_length_in_minutes` - 4 字节
    - `created_at` - 5 字节
    - `paste_path` - 255 字节
    - 总计：大约 1.27 KB
- 每个月的粘贴造作将会产生 12.7 GB 的记录
    - 每次粘贴 1.27 KB * 1000 万次粘贴
    - 3年内大约产生了 450 GB 的新内容记录
    - 3年内生成了 36000 万个短链接
    - 假设大多数的粘贴操作都是新的粘贴而不是更新以前的粘贴内容
- 平均每秒 4 次读取粘贴
- 平均每秒 40 次读取粘贴请求

便利换算指南：

- 每个月有 250 万秒
- 每秒一个请求 = 每个月 250 万次请求
- 每秒 40 个请求 = 每个月 1 亿次请求
- 每秒 400 个请求 = 每个月 10 亿次请求

第二步：概要设计

> 列出所有重要组件以规划概要设计。



第三步：设计核心组件

> 深入每个核心组件的细节。

用例：用户输入一些文本，然后得到一个随机生成的链接

我们将使用关系型数据库，将其作为一个超大哈希表，将生成的 url 和文件服务器上对应文件的路径一一对应。

我们可以使用诸如 Amazon S3 之类的对象存储服务或者NoSQL来代替自建文件服务器。

除了使用关系型数据库来作为一个超大哈希表之外，我们也可以使用NoSQL来代替它。究竟是用 SQL 还是用 NoSQL。不过在下面的讨论中，我们默认选择了使用关系型数据库的方案。

- 客户端向向运行反向代理的 **Web** 服务器发送一个粘贴请求
- **Web** 服务器 将请求转发给**Write API** 服务
- **Write API**服务将会：
    - 生成一个独一无二的 url
        - 通过在 **SQL** 数据库中查重来确认这个 url 是否的确独一无二
        - 如果这个 url 已经存在了，重新生成一个 url
        - 如果支持自定义 url，我们也可以使用用户提供的 url（也需要进行查重）
    - 将 url 存入 **SQL** 数据库的 `pastes` 表中
    - 将粘贴的数据存入对象存储系统中
    - 返回 url

向你的面试官告知你准备写多少代码。

`pastes` 表的数据结构如下：

```
shortlink char(7) NOT NULL
expiration_length_in_minutes int NOT NULL
created_at datetime NOT NULL
paste_path varchar(255) NOT NULL
PRIMARY KEY(shortlink)
```

我们会以 `shortlink` 与 `created_at` 创建一个 索引 以加快查询速度（只需要使用读取日志的时间，不再需要每次都扫描整个数据表）并让数据常驻内存。从内存读取 1 MB 连续数据大约要花 250 微秒，而从 SSD 读取同样大小的数据要花费 4 倍的时间，从机械硬盘读取需要花费 80 倍以上的时间。[1]

为了生成独一无二的 url，我们需要：

- 对用户的 IP 地址 + 时间戳进行 **MD5** 哈希编码
    - MD5 是一种非常常用的哈希化函数，它能生成 128 字节的哈希值
    - MD5 是均匀分布的
    - 另外，我们可以使用 MD5 哈希算法来生成随机数据
- 对 MD5 哈希值进行 **Base 62** 编码
    - Base 62 编码后的值由 `[a-zA-Z0-9]` 组成，它们可以直接作为 url 的字符，不需要再次转义
    - 在这儿仅仅只对原始输入进行过一次哈希处理，Base 62 编码步骤是确定性的（不涉及随机性）
    - Base 64 是另一种很流行的编码形式，但是它生成的字符串作为 url 存在一些问题：

Base 64m字符串内包含 `+` 和 `/` 符号
- 下面的 Base 62 pseudocode 算法时间复杂度为 O(k)，本例中取 num =7，即 k 值为 7：

```
def base_encode(num, base=62):
    digits = []
    while num > 0
      remainder = modulo(num, base)
      digits.push(remainder)
      num = divide(num, base)
    digits = digits.reverse
```

- 输出前 7 个字符，其结果将有 62^7 种可能的值，作为短链接来说足够了。因为我们限制了 3 年内最多产生 36000 万个短链接：

```
url = base_encode(md5(ip_address+timestamp))[:URL_LENGTH]
```

我们可以调用一个公共的 REST API：

```
$ curl -X POST --data '{ "expiration_length_in_minutes": "60", \
    "paste_contents": "Hello World!" }' https://pastebin.com/api/v1/paste
```

返回：

```
{
    "shortlink": "foobar"
}
```

而对于服务器内部的通信，我们可以使用 RPC。

用例：用户输入了一个之前粘贴得到的 url，希望浏览其存储的内容

- 客户端向**Web** 服务器发起读取内容请求
- **Web** 服务器将请求转发给**Read API**服务
- **Read API**服务将会：
  - 在**SQL** 数据库中检查生成的 url
    - 如果查询的 url 存在于 **SQL** 数据库中，从对象存储服务将对应的粘贴内容取出
    - 否则，给用户返回报错

REST API:

```
$ curl https://pastebin.com/api/v1/paste?shortlink=foobar
```

返回：

```
{
    "paste_contents": "Hello World"
    "created_at": "YYYY-MM-DD HH:MM:SS"
    "expiration_length_in_minutes": "60"
}
```

用例：对页面进行跟踪分析

由于不需要进行实时分析，因此我们可以简单地对 **Web** 服务产生的日志用 **MapReduce** 来统计 hit 计数（命中数）。

向你的面试官告知你准备写多少代码。

```
class HitCounts(MRJob):

    def extract_url(self, line):
        """从 log 中取出生成的 url。"""
        ...

    def extract_year_month(self, line):
        """返回时间戳中表示年份与月份的一部分"""
        ...

    def mapper(self, _, line):
        """解析日志的每一行，提取并转换相关行，

        将键值对设定为如下形式：

        (2016-01, url0), 1
        (2016-01, url0), 1
        (2016-01, url1), 1
        """
        url = self.extract_url(line)
        period = self.extract_year_month(line)
        yield (period, url), 1

    def reducer(self, key, value):
        """将所有的 key 加起来

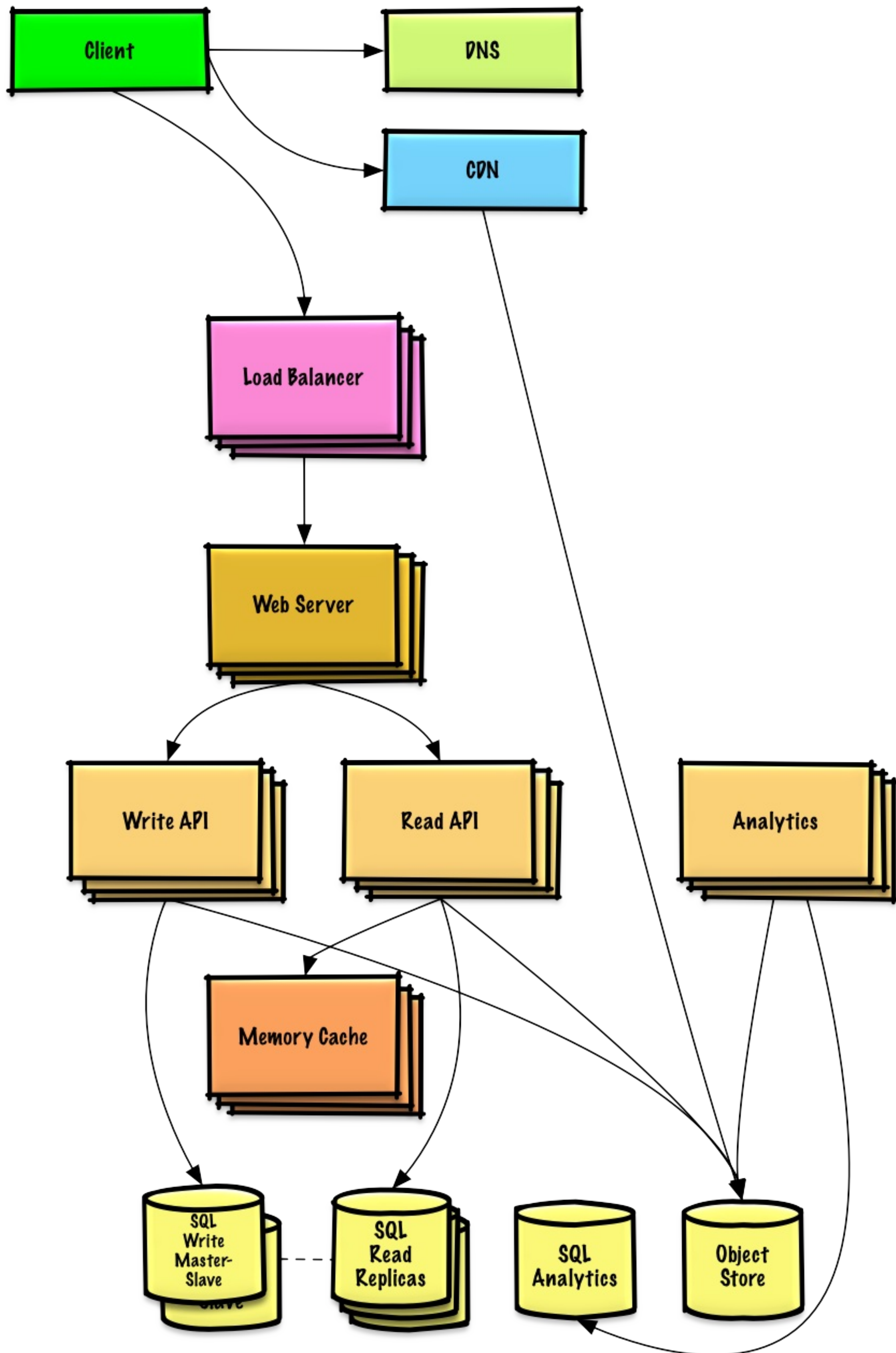
        (2016-01, url0), 2
        (2016-01, url1), 1
        """
        yield key, sum(values)
```

用例：服务删除过期的粘贴内容

我们可以通过扫描 **SQL** 数据库，查找出那些过期时间戳小于当前时间戳的条目，然后在表中删除（或者将其标记为过期）这些过期的粘贴内容。

第四步：架构扩展

> 根据限制条件，找到并解决瓶颈。



重要提示：不要从最初设计直接跳到最终设计中！

现在你要 1) 基准测试、负载测试。2) 分析、描述性能瓶颈。3) 在解决瓶颈问题的同时，评估替代方案、权衡利弊。4) 重复以上步骤。请阅读「设计一个系统，并将其扩大到为数以百万计的 AWS 用户服务」 来了解如何逐步扩大初始设计。

讨论初始设计可能遇到的瓶颈及相关解决方案是很重要的。例如加上一个配置多台 **Web** 服务器的负载均衡器是否能够解决问题？**CDN**呢？主从复制呢？它们各自的替代方案和需要权衡的利弊又有什么呢？

我们将会介绍一些组件来完成设计，并解决架构扩张问题。内置的负载均衡器将不做讨论以节省篇幅。

为了避免重复讨论，请参考系统设计主题索引相关部分来了解其要点、方案的权衡取舍以及可选的替代方案。

- DNS
- 负载均衡器
- 水平拓展
- 反向代理（web 服务器）
- API 服务（应用层）
- 缓存
- 关系型数据库管理系统 (RDBMS)
- SQL 故障主从切换
- 主从复制
- 一致性模式
- 可用性模式

分析数据库 可以用现成的数据仓储系统，例如使用 Amazon Redshift 或者 Google BigQuery 的解决方案。

Amazon S3 的对象存储系统可以很方便地设置每个月限制只允许新增 12.7 GB 的存储内容。

平均每秒 40 次的读取请求（峰值将会更高）, 可以通过扩展 内存缓存 来处理热点内容的读取流量，这对于处理不均匀分布的流量和流量峰值也很有用。只要 SQL 副本不陷入复制-写入困境中，**SQL Read** 副本 基本能够处理缓存命中问题。

平均每秒 4 次的粘贴写入操作（峰值将会更高）对于单个**SQL** 写主**-**从 模式来说是可行的。不过，我们也需要考虑其它的 SQL 性能拓展技术：

- 联合
- 分片
- 非规范化
- SQL 调优

我们也可以考虑将一些数据移至 **NoSQL** 数据库。

其它要点

> 是否深入这些额外的主题，取决于你的问题范围和剩下的时间。

NoSQL

- 键-值存储
- 文档类型存储
- 列型存储
- 图数据库
- SQL vs NoSQL

缓存

- 在哪缓存
    - 客户端缓存
    - CDN 缓存
    - Web 服务器缓存
    - 数据库缓存
    - 应用缓存
- 什么需要缓存
    - 数据库查询级别的缓存
    - 对象级别的缓存
- 何时更新缓存
    - 缓存模式
    - 直写模式
    - 回写模式
    - 刷新

异步与微服务

- 消息队列
- 任务队列
- 背压
- 微服务

通信

- 可权衡选择的方案：
    - 与客户端的外部通信 - 使用 REST 作为 HTTP API

    - 服务器内部通信 - RPC
- 服务发现

安全性

请参阅「安全」一章。

延迟数值

请参阅「每个程序员都应该知道的延迟数」。

持续探讨

- 持续进行基准测试并监控你的系统，以解决他们提出的瓶颈问题。
- 架构拓展是一个迭代的过程。

设计推特时间轴与搜索功能

注意：这个文档中的链接会直接指向系统设计主题索引中的有关部分，以避免重复的内容。你可以参考链接的相关内容，来了解其总的要点、方案的权衡取舍以及可选的替代方案。

设计 **Facebook** 的 **feed** 与设计 **Facebook** 搜索与此为同一类型问题。

第一步：简述用例与约束条件

> 搜集需求与问题的范围。提出问题来明确用例与约束条件。讨论假设。

我们将在没有面试官明确说明问题的情况下，自己定义一些用例以及限制条件。

用例

我们将把问题限定在仅处理以下用例的范围中

- 用户发布了一篇推特
  - 服务将推特推送给关注者，给他们发送消息通知与邮件
- 用户浏览用户时间轴（用户最近的活动）
- 用户浏览主页时间轴（用户关注的人最近的活动）
- 用户搜索关键词
- 服务需要有高可用性

不在用例范围内的有

- 服务向 Firehose 与其它流数据接口推送推特
- 服务根据用户的"是否可见"选项排除推特
  - 隐藏未关注者的 @回复
  - 关心"隐藏转发"设置
- 数据分析

限制条件与假设

提出假设

普遍情况

- 网络流量不是均匀分布的

- 发布推特的速度需要足够快速
    - 除非有上百万的关注者，否则将推特推送给粉丝的速度要足够快
- 1 亿个活跃用户
- 每天新发布 5 亿条推特，每月新发布 150 亿条推特
    - 平均每条推特需要推送给 5 个人
    - 每天需要进行 50 亿次推送
    - 每月需要进行 1500 亿次推送
- 每月需要处理 2500 亿次读取请求
- 每月需要处理 100 亿次搜索

时间轴功能

- 浏览时间轴需要足够快
- 推特的读取负载要大于写入负载
    - 需要为推特的快速读取进行优化
- 存入推特是高写入负载功能

搜索功能

- 搜索速度需要足够快
- 搜索是高负载读取功能

计算用量

如果你需要进行粗略的用量计算，请向你的面试官说明。

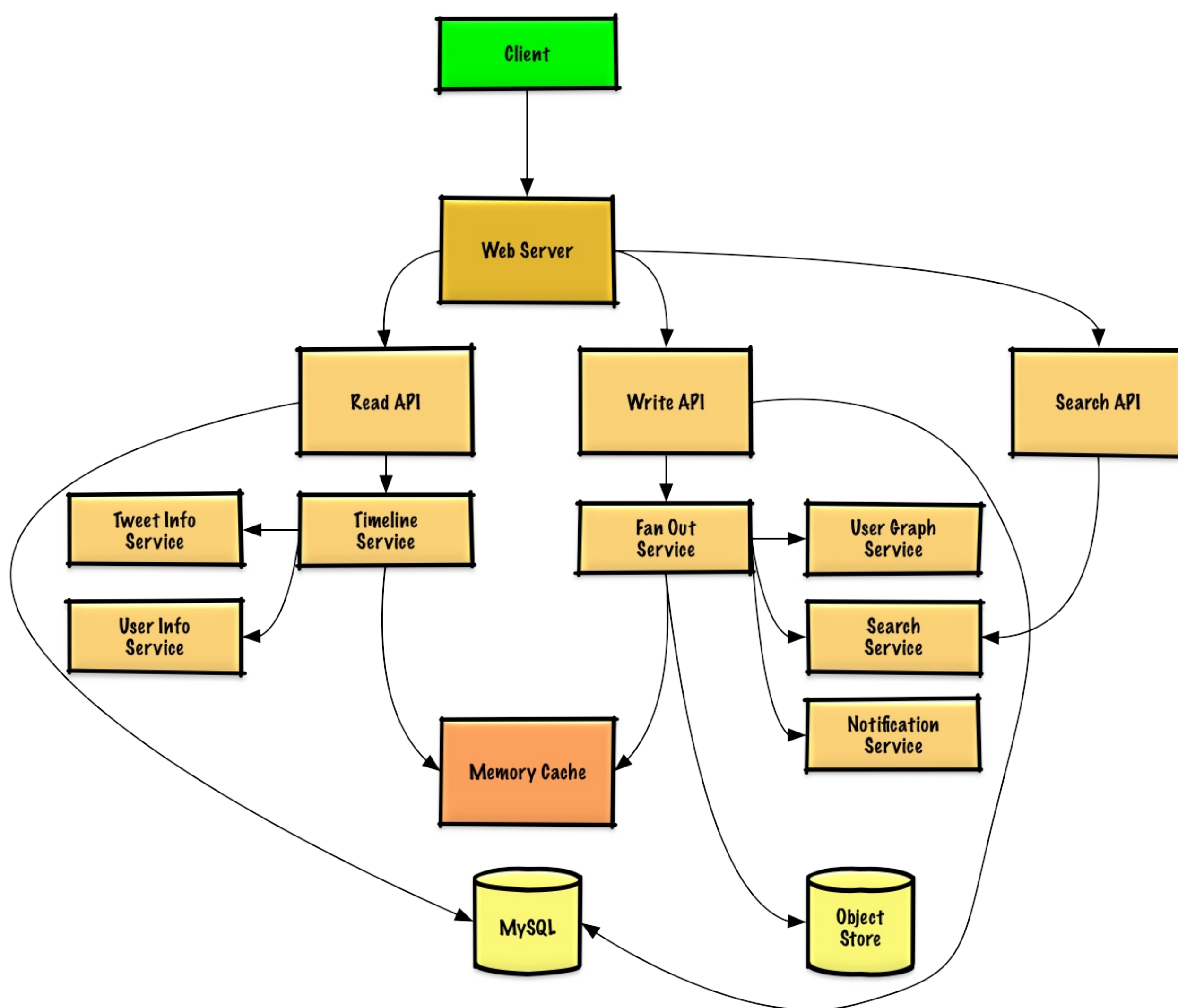
- 每条推特的大小：
    - `tweet_id` - 8 字节
    - `user_id` - 32 字节
    - `text` - 140 字节
    - `media` - 平均 10 KB
    - 总计： 大约 10 KB
- 每月产生新推特的内容为 150 TB
    - 每条推特 10 KB 每天 5 亿条推特 每月 30 天
    - 3 年产生新推特的内容为 5.4 PB
- 每秒需要处理 10 万次读取请求
    - 每个月需要处理 2500 亿次请求 * (每秒 400 次请求 / 每月 10 亿次请求)
- 每秒发布 6000 条推特
    - 每月发布 150 亿条推特 * (每秒 400 次请求 / 每月 10 次请求)
- 每秒推送 6 万条推特
    - 每月推送 1500 亿条推特 * (每秒 400 次请求 / 每月 10 亿次请求)
- 每秒 4000 次搜索请求

便利换算指南：

- 每个月有 250 万秒
- 每秒一个请求 = 每个月 250 万次请求
- 每秒 40 个请求 = 每个月 1 亿次请求
- 每秒 400 个请求 = 每个月 10 亿请求

第二步：概要设计

> 列出所有重要组件以规划概要设计。



第三步：设计核心组件

> 深入每个核心组件的细节。

用例：用户发表了一篇推特

我们可以将用户自己发表的推特存储在关系数据库中。我们也可以讨论一下究竟是用 SQL 还是用 NoSQL。

构建用户主页时间轴（查看关注用户的活动）以及推送推特是件麻烦事。将特推传播给所有关注者（每秒约递送 6 万条推特）这一操作有可能会使传统的关系数据库超负载。因此，我们可以使用 **NoSQL** 数据库或内存数据库之类的更快的数据存储方式。从内存读取 1 MB 连续数据大约要花 250 微秒，而从 SSD 读取同样大小的数据要花费 4 倍的时间，从机械硬盘读取需要花费 80 倍以上的时间。[1]

我们可以将照片、视频之类的媒体存储于对象存储中。

- 客户端向应用反向代理的**Web** 服务器发送一条推特
- **Web** 服务器将请求转发给写 **API**服务器
- 写 **API**服务器将推特使用 **SQL** 数据库存储于用户时间轴中
- 写 **API**调用消息输出服务，进行以下操作：
  - 查询用户 图 服务找到存储于内存缓存中的此用户的粉丝
  - 将推特存储于内存缓存中的此用户的粉丝的主页时间轴中
    - O(n) 复杂度操作：1000 名粉丝 = 1000 次查找与插入
  - 将特推存储在搜索索引服务中，以加快搜索
  - 将媒体存储于对象存储中
  - 使用通知服务向粉丝发送推送：
    - 使用队列异步推送通知

向你的面试官告知你准备写多少代码。

如果我们用 Redis 作为内存缓存，那可以用 Redis 原生的 list 作为其数据结构。结构如下：

```
           tweet n+2                   tweet n+1                   tweet n
| 8 bytes   8 bytes  1 byte | 8 bytes   8 bytes  1 byte | 8 bytes   7 bytes  1 byte |
| tweet_id  user_id  meta   | tweet_id  user_id  meta   | tweet_id  user_id  meta   |
```

新发布的推特将被存储在对应用户（关注且活跃的用户）的主页时间轴的内存缓存中。

我们可以调用一个公共的 REST API：

```
$ curl -X POST --data '{ "user_id": "123", "auth_token": "ABC123", \
    "status": "hello world!", "media_ids": "ABC987" }' \
    https://twitter.com/api/v1/tweet
```

返回：

```
{
    "created_at": "Wed Sep 05 00:37:15 +0000 2012",
    "status": "hello world!",
    "tweet_id": "987",
    "user_id": "123",
    ...
}
```

而对于服务器内部的通信，我们可以使用 RPC。

用例：用户浏览主页时间轴

- 客户端向 **Web** 服务器发起一次读取主页时间轴的请求
- **Web** 服务器将请求转发给读取 **API**服务器
- 读取 **API**服务器调用时间轴服务进行以下操作：
    - 从内存缓存读取时间轴数据，其中包括推特 id 与用户 id - O(1)
    - 通过 multiget 向推特信息服务进行查询，以获取相关 id 推特的额外信息 - O(n)
    - 通过 muiltiget 向用户信息服务进行查询，以获取相关 id 用户的额外信息 - O(n)

REST API：

```
$ curl https://twitter.com/api/v1/home_timeline?user_id=123
```

返回：

```
{
    "user_id": "456",
    "tweet_id": "123",
    "status": "foo"
},
{
    "user_id": "789",
    "tweet_id": "456",
    "status": "bar"
},
{
    "user_id": "789",
    "tweet_id": "579",
    "status": "baz"
},
```

用例：用户浏览用户时间轴

- 客户端向**Web** 服务器发起获得主页时间轴的请求
- **Web** 服务器将请求转发给读取 **API**服务器

* 读取 **API**从 **SQL** 数据库中取出用户的时间轴

REST API 与前面的主页时间轴类似，区别只在于取出的推特是由用户自己发送而不是关注人发送。

用例：用户搜索关键词

* 客户端将搜索请求发给**Web** 服务器
* **Web** 服务器将请求转发给搜索 **API**服务器
* 搜索 **API**调用搜索服务进行以下操作：
  * 对输入进行转换与分词，弄明白需要搜索什么东西
    * 移除标点等额外内容
    * 将文本打散为词组
    * 修正拼写错误
    * 规范字母大小写
    * 将查询转换为布尔操作
  * 查询搜索集群（例如Lucene）检索结果：
    * 对集群内的所有服务器进行查询，将有结果的查询进行发散聚合（Scatter gathers）
    * 合并取到的条目，进行评分与排序，最终返回结果

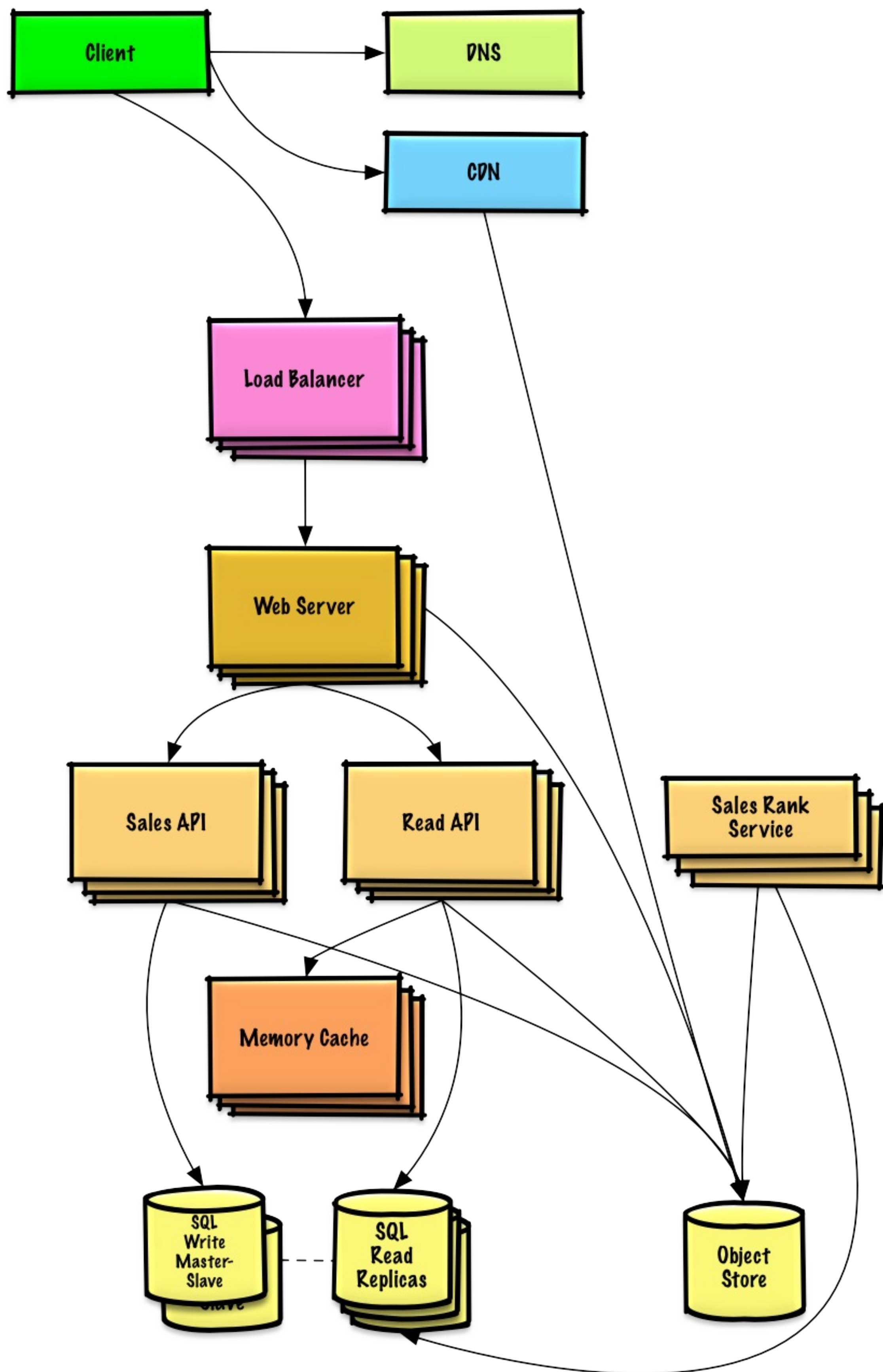
REST API：

```
$ curl https://twitter.com/api/v1/search?query=hello+world
```

返回结果与前面的主页时间轴类似，只不过返回的是符合查询条件的推特。

第四步：架构扩展

> 根据限制条件，找到并解决瓶颈。



重要提示：不要从最初设计直接跳到最终设计中！

现在你要 1) 基准测试、负载测试。2) 分析、描述性能瓶颈。3) 在解决瓶颈问题的同时，评估替代方案、权衡利弊。4) 重复以上步骤。请阅读「设计一个系统，并将其扩大到为数以百万计的 AWS 用户服务」 来了解如何逐步扩大初始设计。

讨论初始设计可能遇到的瓶颈及相关解决方案是很重要的。例如加上一个配置多台 **Web** 服务器的负载均衡器是否能够解决问题？**CDN**呢？主从复制呢？它们各自的替代方案和需要权衡的利弊又有什么呢？

我们将会介绍一些组件来完成设计，并解决架构扩张问题。内置的负载均衡器将不做讨论以节省篇幅。

为了避免重复讨论，请参考系统设计主题索引相关部分来了解其要点、方案的权衡取舍以及可选的替代方案。

- DNS
- 负载均衡器
- 水平拓展
- 反向代理（web 服务器）
- API 服务（应用层）
- 缓存
- 关系型数据库管理系统 (RDBMS)
- SQL 故障主从切换
- 主从复制
- 一致性模式
- 可用性模式

消息输出服务有可能成为性能瓶颈。那些有着百万数量关注着的用户可能发一条推特就需要好几分钟才能完成消息输出进程。这有可能使 @回复 这种推特时出现竞争条件，因此需要根据服务时间对此推特进行重排序来降低影响。

我们还可以避免从高关注量的用户输出推特。相反，我们可以通过搜索来找到高关注量用户的推特，并将搜索结果与用户的主页时间轴合并，再根据时间对其进行排序。

此外，还可以通过以下内容进行优化：

- 仅为每个主页时间轴在内存缓存中存储数百条推特
- 仅在内存缓存中存储活动用户的主页时间轴
  - 如果某个用户在过去 30 天都没有产生活动，那我们可以使用 **SQL** 数据库重新构建他的时间轴
    - 使用用户 图 服务来查询并确定用户关注的人
    - 从 **SQL** 数据库中取出推特，并将它们存入内存缓存
- 仅在推特信息服务中存储一个月的推特
- 仅在用户信息服务中存储活动用户的信息
- 搜索集群需要将推特保留在内存中，以降低延迟

我们还可以考虑优化 **SQL 数据库** 来解决一些瓶颈问题。

内存缓存能减小一些数据库的负载，靠 **SQL Read** 副本已经足够处理缓存未命中情况。我们还可以考虑使用一些额外的 SQL 性能拓展技术。

高容量的写入将淹没单个的 **SQL 写主从模式**，因此需要更多的拓展技术。

- 联合
- 分片
- 非规范化
- SQL 调优

我们也可以考虑将一些数据移至 **NoSQL 数据库**。

其它要点

> 是否深入这些额外的主题，取决于你的问题范围和剩下的时间。

NoSQL

- 键-值存储
- 文档类型存储
- 列型存储
- 图数据库
- SQL vs NoSQL

缓存

- 在哪缓存
  - 客户端缓存
  - CDN 缓存
  - Web 服务器缓存
  - 数据库缓存
  - 应用缓存
- 什么需要缓存
  - 数据库查询级别的缓存
  - 对象级别的缓存
- 何时更新缓存
  - 缓存模式
  - 直写模式
  - 回写模式
  - 刷新

异步与微服务

- 消息队列
- 任务队列
- 背压
- 微服务

通信

- 可权衡选择的方案：
  - 与客户端的外部通信 - 使用 REST 作为 HTTP API
  - 服务器内部通信 - RPC
- 服务发现

安全性

请参阅「安全」一章。

延迟数值

请参阅「每个程序员都应该知道的延迟数」。

持续探讨

- 持续进行基准测试并监控你的系统，以解决他们提出的瓶颈问题。
- 架构拓展是一个迭代的过程。

设计一个网页爬虫

注意：这个文档中的链接会直接指向系统设计主题索引中的有关部分，以避免重复的内容。你可以参考链接的相关内容，来了解其总的要点、方案的权衡取舍以及可选的替代方案。

第一步：简述用例与约束条件

> 把所有需要的东西聚集在一起，审视问题。不停的提问，以至于我们可以明确使用场景和约束。讨论假设。

我们将在没有面试官明确说明问题的情况下，自己定义一些用例以及限制条件。

用例

我们把问题限定在仅处理以下用例的范围中

- 服务 抓取一系列链接：
  - 生成包含搜索词的网页倒排索引
  - 生成页面的标题和摘要信息
    - 页面标题和摘要都是静态的，它们不会根据搜索词改变
- 用户 输入搜索词后，可以看到相关的搜索结果列表，列表每一项都包含由网页爬虫生成的页面标题及摘要
  - 只给该用例绘制出概要组件和交互说明，无需讨论细节
- 服务 具有高可用性

无需考虑

- 搜索分析
- 个性化搜索结果
- 页面排名

限制条件与假设

提出假设

- 搜索流量分布不均
  - 有些搜索词非常热门，有些则非常冷门
- 只支持匿名用户

- 用户很快就能看到搜索结果
- 网页爬虫不应该陷入死循环
    - 当爬虫路径包含环的时候，将会陷入死循环
- 抓取 10 亿个链接
    - 要定期重新抓取页面以确保新鲜度
    - 平均每周重新抓取一次，网站越热门，那么重新抓取的频率越高
        - 每月抓取 40 亿个链接
    - 每个页面的平均存储大小：500 KB
        - 简单起见，重新抓取的页面算作新页面
- 每月搜索量 1000 亿次

用更传统的系统来练习 —— 不要使用 solr 、nutch 之类的现成系统。

计算用量

如果你需要进行粗略的用量计算，请向你的面试官说明。

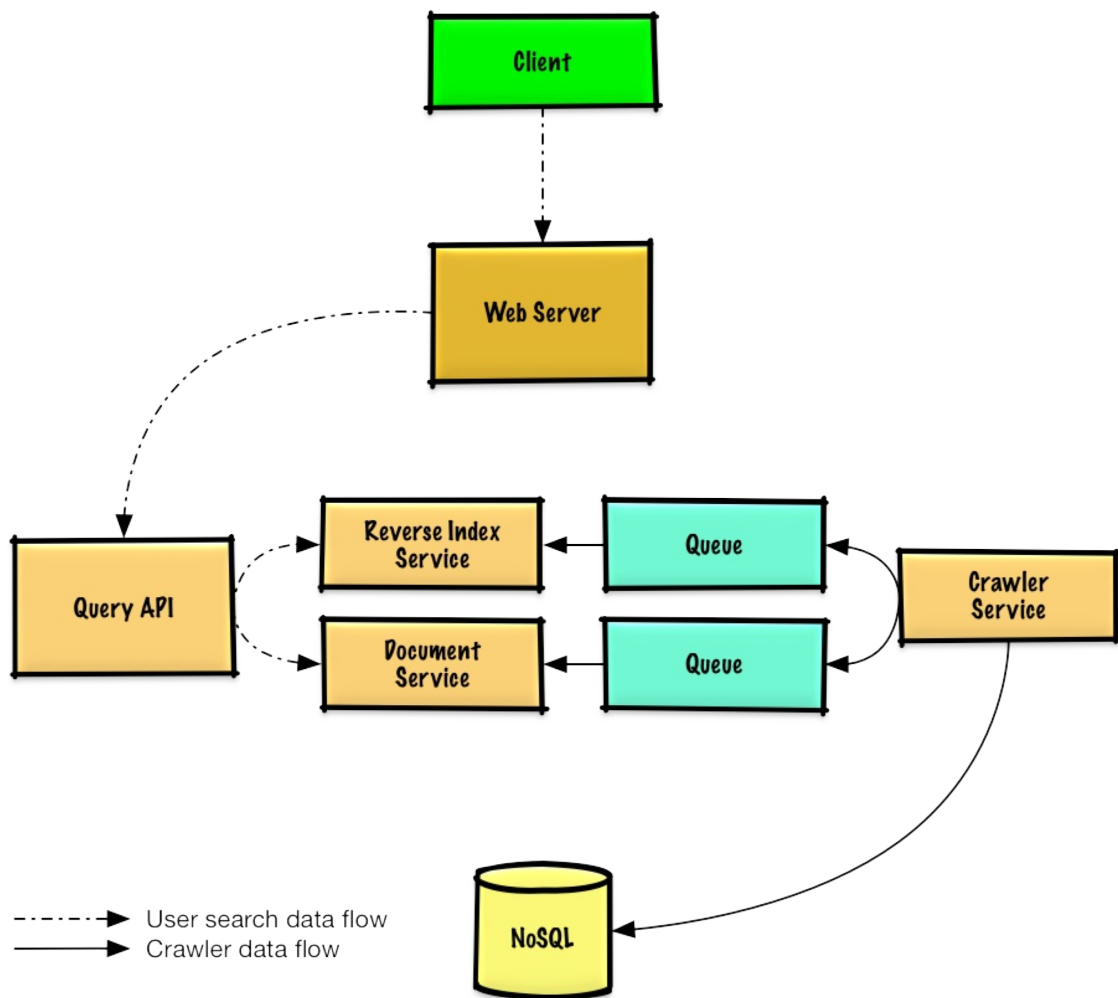
- 每月存储 2 PB 页面
    - 每月抓取 40 亿个页面，每个页面 500 KB
    - 三年存储 72 PB 页面
- 每秒 1600 次写请求
- 每秒 40000 次搜索请求

简便换算指南：

- 一个月有 250 万秒
- 每秒 1 个请求，即每月 250 万个请求
- 每秒 40 个请求，即每月 1 亿个请求
- 每秒 400 个请求，即每月 10 亿个请求

第二步：概要设计

> 列出所有重要组件以规划概要设计。



第三步：设计核心组件

> 对每一个核心组件进行详细深入的分析。

用例：爬虫服务抓取一系列网页

假设我们有一个初始列表 `links_to_crawl`（待抓取链接），它最初基于网站整体的知名度来排序。当然如果这个假设不合理，我们可以使用 Yahoo、DMOZ 等知名门户网站作为种子链接来进行扩散 。

我们将用表 `crawled_links` （已抓取链接 ）来记录已经处理过的链接以及相应的页面签名。

我们可以将 `links_to_crawl` 和 `crawled_links` 记录在键-值型 **NoSQL** 数据库中。对于 `crawled_links` 中已排序的链接，我们可以使用 Redis 的有序集合来维护网页链接的排名。我们应当在 选择 SQL 还是 NoSQL 的问题上，讨论有关使用场景以及利弊 。

- 爬虫服务按照以下流程循环处理每一个页面链接：
  - 选取排名最靠前的待抓取链接
    - 在 NoSQL 数据库的 `crawled_links` 中，检查待抓取页面的签名是否与某个已抓取页面的签名相似
      - 若存在，则降低该页面链接的优先级
        - 这样做可以避免陷入死循环
        - 继续（进入下一次循环）
      - 若不存在，则抓取该链接
        - 在倒排索引服务任务队列中，新增一个生成倒排索引任务。
        - 在文档服务任务队列中，新增一个生成静态标题和摘要的任务。
        - 生成页面签名
        - 在 NoSQL 数据库的 `links_to_crawl` 中删除该链接
        - 在 NoSQL 数据库的 `crawled_links` 中插入该链接以及页面签名

向面试官了解你需要写多少代码。

`PagesDataStore` 是爬虫服务中的一个抽象类，它使用 NoSQL 数据库进行存储。

```
class PagesDataStore(object):

    def __init__(self, db);
        self.db = db
        ...

    def add_link_to_crawl(self, url):
        """将指定链接加入 `links_to_crawl`。"""
        ...

    def remove_link_to_crawl(self, url):
        """从 `links_to_crawl` 中删除指定链接。"""
        ...

    def reduce_priority_link_to_crawl(self, url)
        """在 `links_to_crawl` 中降低一个链接的优先级以避免死循环。"""
        ...

    def extract_max_priority_page(self):
        """返回 `links_to_crawl` 中优先级最高的链接。"""
        ...

    def insert_crawled_link(self, url, signature):
        """将指定链接加入 `crawled_links`。"""
        ...

    def crawled_similar(self, signature):
        """判断待抓取页面的签名是否与某个已抓取页面的签名相似。"""
        ...
```

`Page` 是爬虫服务的一个抽象类，它封装了网页对象，由页面链接、页面内容、子链接和页面签名构成。

```
class Page(object):

    def __init__(self, url, contents, child_urls, signature):
        self.url = url
        self.contents = contents
        self.child_urls = child_urls
        self.signature = signature
```

`Crawler` 是爬虫服务的主类，由`Page` 和 `PagesDataStore` 组成。

```
class Crawler(object):

    def __init__(self, data_store, reverse_index_queue, doc_index_queue):
        self.data_store = data_store
        self.reverse_index_queue = reverse_index_queue
        self.doc_index_queue = doc_index_queue

    def create_signature(self, page):
        """基于页面链接与内容生成签名。"""
        ...

    def crawl_page(self, page):
        for url in page.child_urls:
            self.data_store.add_link_to_crawl(url)
        page.signature = self.create_signature(page)
        self.data_store.remove_link_to_crawl(page.url)
        self.data_store.insert_crawled_link(page.url, page.signature)

    def crawl(self):
        while True:
            page = self.data_store.extract_max_priority_page()
            if page is None:
                break
            if self.data_store.crawled_similar(page.signature):
                self.data_store.reduce_priority_link_to_crawl(page.url)
            else:
                self.crawl_page(page)
```

处理重复内容

我们要谨防网页爬虫陷入死循环，这通常会发生在爬虫路径中存在环的情况。

向面试官了解你需要写多少代码.

删除重复链接：

* 假设数据量较小，我们可以用类似于 `sort | unique` 的方法。（译注： 先排序，后去重）
* 假设有 10 亿条数据，我们应该使用 **MapReduce** 来输出只出现 1 次的记录。

```
class RemoveDuplicateUrls(MRJob):

    def mapper(self, _, line):
        yield line, 1

    def reducer(self, key, values):
        total = sum(values)
        if total == 1:
            yield key, total
```

比起处理重复内容，检测重复内容更为复杂。我们可以基于网页内容生成签名，然后对比两者签名的相似度。可能会用到的算法有 Jaccard index 以及 cosine similarity。

抓取结果更新策略

要定期重新抓取页面以确保新鲜度。抓取结果应该有个 `timestamp` 字段记录上一次页面抓取时间。每隔一段时间，比如说 1 周，所有页面都需要更新一次。对于热门网站或是内容频繁更新的网站，爬虫抓取间隔可以缩短。

尽管我们不会深入网页数据分析的细节，我们仍然要做一些数据挖掘工作来确定一个页面的平均更新时间，并且根据相关的统计数据来决定爬虫的重新抓取频率。

当然我们也应该根据站长提供的 `Robots.txt` 来控制爬虫的抓取频率。

用例：用户输入搜索词后，可以看到相关的搜索结果列表，列表每一项都包含由网页爬虫生成的页面标题及摘要

* 客户端向运行反向代理的 **Web** 服务器发送一个请求
* **Web** 服务器 发送请求到 **Query API** 服务器
* 查询 **API** 服务将会做这些事情：
    * 解析查询参数
        * 删除 HTML 标记
        * 将文本分割成词组 （译注： 分词处理）
        * 修正错别字
        * 规范化大小写
        * 将搜索词转换为布尔运算
    * 使用倒排索引服务来查找匹配查询的文档
        * 倒排索引服务对匹配到的结果进行排名，然后返回最符合的结果
    * 使用文档服务返回文章标题与摘要

我们使用 **REST API** 与客户端通信：

```
$ curl https://search.com/api/v1/search?query=hello+world
```

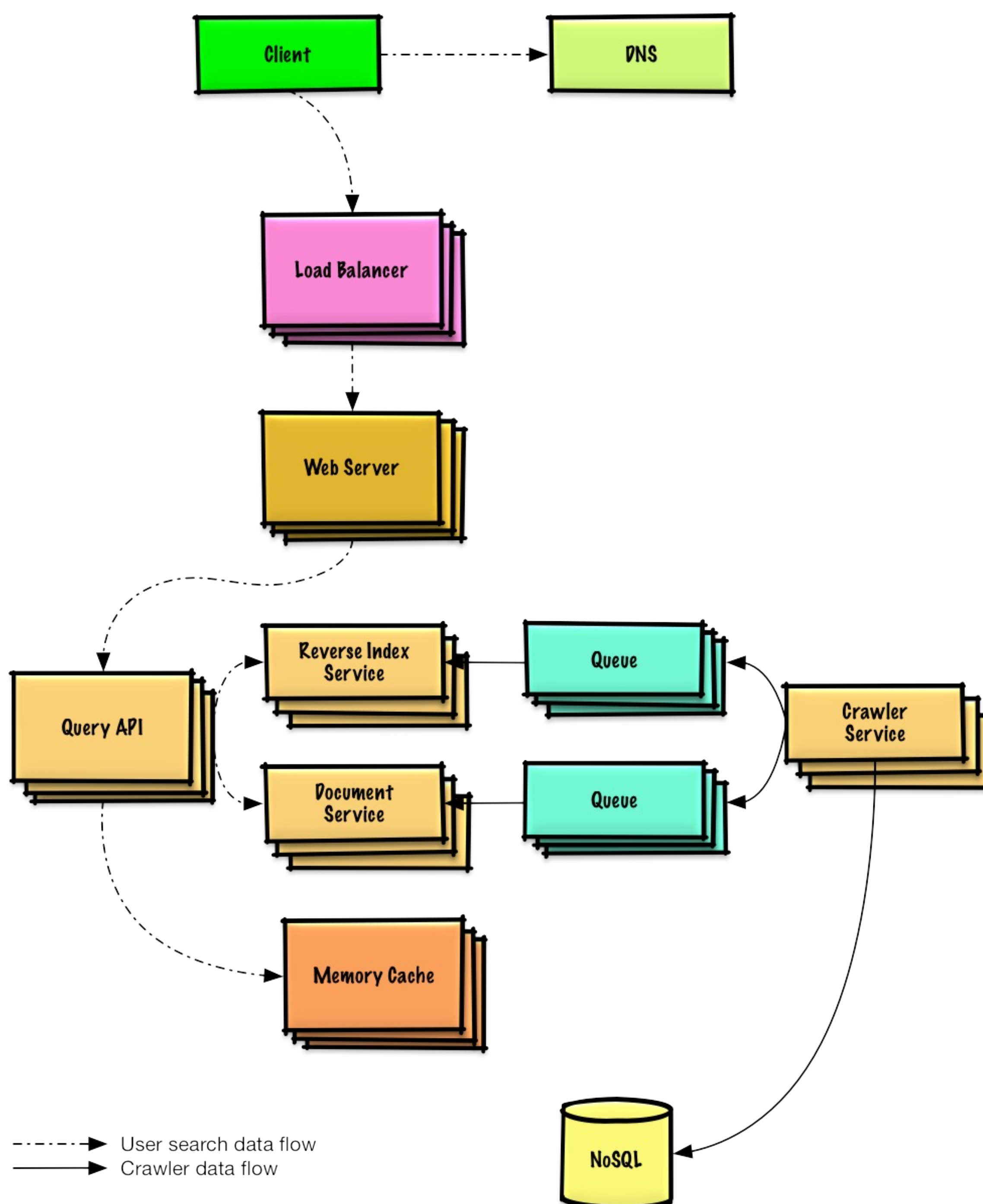
响应内容:

```
{
    "title": "foo's title",
    "snippet": "foo's snippet",
    "link": "https://foo.com",
},
{
    "title": "bar's title",
    "snippet": "bar's snippet",
    "link": "https://bar.com",
},
{
    "title": "baz's title",
    "snippet": "baz's snippet",
    "link": "https://baz.com",
},
```

对于服务器内部通信，我们可以使用 远程过程调用协议（RPC）

第四步：架构扩展

> 根据限制条件，找到并解决瓶颈。



重要提示：不要直接从最初设计跳到最终设计！

现在你要 1) 基准测试、负载测试。2) 分析、描述性能瓶颈。3) 在解决瓶颈问题的同时，评估替代方案、权衡利弊。4) 重复以上步骤。请阅读设计一个系统，并将其扩大到为数以百万计的 AWS 用户服务 来了解如何逐步扩大初始设计。

讨论初始设计可能遇到的瓶颈及相关解决方案是很重要的。例如加上一套配备多台 **Web** 服务器的负载均衡器是否能够解决问题？**CDN**呢？主从复制呢？它们各自的替代方案和需要权衡的利弊又有哪些呢？

我们将会介绍一些组件来完成设计，并解决架构规模扩张问题。内置的负载均衡器将不做讨论以节省篇幅。

为了避免重复讨论，请参考系统设计主题索引相关部分来了解其要点、方案的权衡取舍以及替代方案。

- DNS
- 负载均衡器
- 水平扩展
- Web 服务器（反向代理）
- API 服务器（应用层）
- 缓存
- NoSQL
- 一致性模式
- 可用性模式

有些搜索词非常热门，有些则非常冷门。热门的搜索词可以通过诸如 Redis 或者 Memcached 之类的内存缓存来缩短响应时间，避免倒排索引服务以及文档服务过载。内存缓存同样适用于流量分布不均匀以及流量短时高峰问题。从内存中读取 1 MB 连续数据大约需要 250 微秒，而从 SSD 读取同样大小的数据要花费 4 倍的时间，从机械硬盘读取需要花费 80 倍以上的时间。[1]

以下是优化爬虫服务的其他建议：

- 为了处理数据大小问题以及网络请求负载，倒排索引服务和文档服务可能需要大量应用数据分片和数据复制。
- DNS 查询可能会成为瓶颈，爬虫服务最好专门维护一套定期更新的 DNS 查询服务。
- 借助于连接池，即同时维持多个开放网络连接，可以提升爬虫服务的性能并减少内存使用量。
    - 改用 UDP 协议同样可以提升性能
- 网络爬虫受带宽影响较大，请确保带宽足够维持高吞吐量。

其它要点

> 是否深入这些额外的主题，取决于你的问题范围和剩下的时间。

SQL 扩展模式

- 读取复制
- 联合
- 分片
- 非规范化

- SQL 调优

NoSQL

- 键-值存储
- 文档类型存储
- 列型存储
- 图数据库
- SQL vs NoSQL

缓存

- 在哪缓存
  - 客户端缓存
  - CDN 缓存
  - Web 服务器缓存
  - 数据库缓存
  - 应用缓存
- 什么需要缓存
  - 数据库查询级别的缓存
  - 对象级别的缓存
- 何时更新缓存
  - 缓存模式
  - 直写模式
  - 回写模式
  - 刷新

异步与微服务

- 消息队列
- 任务队列
- 背压
- 微服务

通信

- 可权衡选择的方案：
  - 与客户端的外部通信 - 使用 REST 作为 HTTP API
  - 内部通信 - RPC
- 服务发现

安全性

请参阅安全。

延迟数值

请参阅每个程序员都应该知道的延迟数。

持续探讨

- 持续进行基准测试并监控你的系统，以解决他们提出的瓶颈问题。
- 架构扩展是一个迭代的过程。

设计 Mint.com

注意：这个文档中的链接会直接指向系统设计主题索引中的有关部分，以避免重复的内容。您可以参考链接的相关内容，来了解其总的要点、方案的权衡取舍以及可选的替代方案。

第一步：简述用例与约束条件

> 搜集需求与问题的范围。提出问题来明确用例与约束条件。讨论假设。

我们将在没有面试官明确说明问题的情况下，自己定义一些用例以及限制条件。

用例

我们将把问题限定在仅处理以下用例的范围中

- 用户 连接到一个财务账户
- 服务 从账户中提取交易
    - 每日更新
    - 分类交易
        - 允许用户手动分类
        - 不自动重新分类
    - 按类别分析每月支出
- 服务 推荐预算
    - 允许用户手动设置预算
    - 当接近或者超出预算时，发送通知
- 服务 具有高可用性

非用例范围

- 服务 执行附加的日志记录和分析

限制条件与假设

提出假设

- 网络流量非均匀分布
- 自动账户日更新只适用于 30 天内活跃的用户
- 添加或者移除财务账户相对较少

- 预算通知不需要及时
- 1000 万用户
  - 每个用户10个预算类别= 1亿个预算项
  - 示例类别:
    - Housing = $1,000
    - Food = $200
    - Gas = $100
  - 卖方确定交易类别
    - 50000 个卖方
- 3000 万财务账户
- 每月 50 亿交易
- 每月 5 亿读请求
- 10:1 读写比
  - Write-heavy，用户每天都进行交易，但是每天很少访问该网站

计算用量

如果你需要进行粗略的用量计算，请向你的面试官说明。

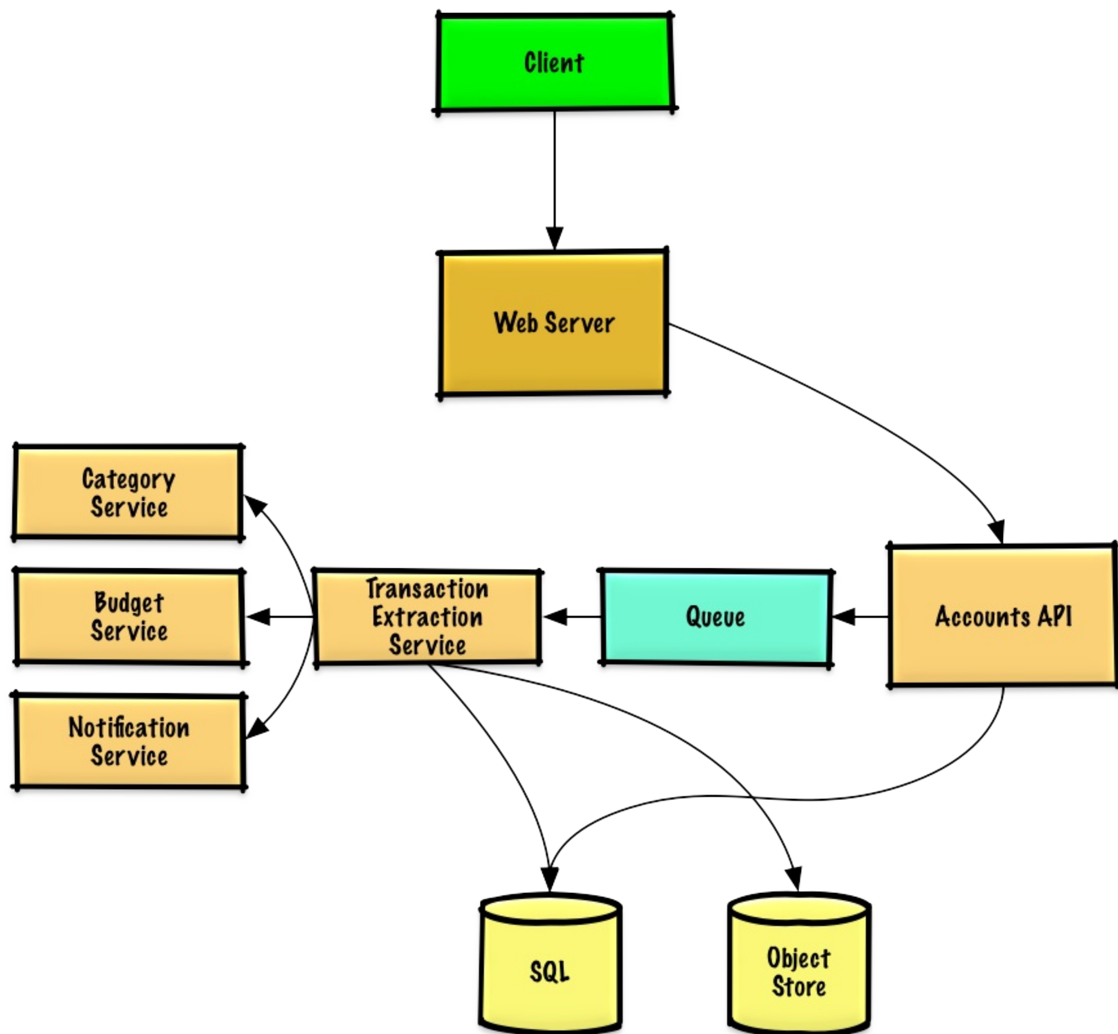
- 每次交易的用量:
  - `user_id` - 8 字节
  - `created_at` - 5 字节
  - `seller` - 32 字节
  - `amount` - 5 字节
  - Total: ~50 字节
- 每月产生 250 GB 新的交易内容
  - 每次交易 50 比特 * 50 亿交易每月
  - 3年内新的交易内容 9 TB
  - Assume most are new transactions instead of updates to existing ones
- 平均每秒产生 2000 次交易
- 平均每秒产生 200 读请求

便利换算指南：

- 每个月有 250 万秒
- 每秒一个请求 = 每个月 250 万次请求
- 每秒 40 个请求 = 每个月 1 亿次请求
- 每秒 400 个请求 = 每个月 10 亿次请求

第二步：概要设计

> 列出所有重要组件以规划概要设计。



第三步：设计核心组件

> 深入每个核心组件的细节。

用例：用户连接到一个财务账户

我们可以将 1000 万用户的信息存储在一个关系数据库中。我们应该讨论一下选择SQL或NoSQL之间的用例和权衡了。

- 客户端 作为一个反向代理，发送请求到 **Web** 服务器
- **Web** 服务器 转发请求到 账户**API** 服务器
- 账户**API** 服务器将新输入的账户信息更新到 **SQL**数据库 的 `accounts` 表

告知你的面试官你准备写多少代码。

`accounts` 表应该具有如下结构：

```
id int NOT NULL AUTO_INCREMENT
created_at datetime NOT NULL
last_update datetime NOT NULL
account_url varchar(255) NOT NULL
account_login varchar(32) NOT NULL
account_password_hash char(64) NOT NULL
user_id int NOT NULL
PRIMARY KEY(id)
FOREIGN KEY(user_id) REFERENCES users(id)
```

我们将在 `id` ，`user_id` 和 `created_at` 等字段上创建一个索引以加速查找（对数时间而不是扫描整个表）并保持数据在内存中。从内存中顺序读取 1 MB数据花费大约250毫秒，而从SSD读取是其4倍，从磁盘读取是其80倍。[1]

我们将使用公开的**REST API**:

```
$ curl -X POST --data '{ "user_id": "foo", "account_url": "bar", \
    "account_login": "baz", "account_password": "qux" }' \
    https://mint.com/api/v1/account
```

对于内部通信，我们可以使用远程过程调用。

接下来，服务从账户中提取交易。

用例：服务从账户中提取交易

如下几种情况下，我们会想要从账户中提取信息：

- 用户首次链接账户
- 用户手动更新账户
- 为过去 30 天内活跃的用户自动日更新

数据流:

- 客户端向 **Web**服务器 发送请求
- **Web**服务器 将请求转发到 帐户**API** 服务器
- 帐户**API** 服务器将job放在 队列 中，如 Amazon SQS 或者 RabbitMQ
    - 提取交易可能需要一段时间，我们可能希望与队列异步地来做，虽然这会引入额外的复杂度。
- 交易提取服务 执行如下操作：
    - 从 **Queue** 中拉取并从金融机构中提取给定用户的交易，将结果作为原始日志文件存储在 对象存储区。
    - 使用 分类服务 来分类每个交易

    - 使用 预算服务 来按类别计算每月总支出
        - 预算服务 使用 通知服务 让用户知道他们是否接近或者已经超出预算
    - 更新具有分类交易的 **SQL**数据库 的 `transactions` 表
    - 按类别更新 **SQL**数据库 `monthly_spending` 表的每月总支出
    - 通过 通知服务 提醒用户交易完成
        - 使用一个 队列 (没有画出来) 来异步发送通知

`transactions` 表应该具有如下结构：

```
id int NOT NULL AUTO_INCREMENT
created_at datetime NOT NULL
seller varchar(32) NOT NULL
amount decimal NOT NULL
user_id int NOT NULL
PRIMARY KEY(id)
FOREIGN KEY(user_id) REFERENCES users(id)
```

我们将在 `id` ， `user_id` ，和 `created_at` 字段上创建索引。

`monthly_spending` 表应该具有如下结构：

```
id int NOT NULL AUTO_INCREMENT
month_year date NOT NULL
category varchar(32)
amount decimal NOT NULL
user_id int NOT NULL
PRIMARY KEY(id)
FOREIGN KEY(user_id) REFERENCES users(id)
```

我们将在 `id` ， `user_id` 字段上创建索引。

分类服务

对于 分类服务，我们可以生成一个带有最受欢迎卖家的卖家-类别字典。如果我们估计 50000 个卖家，并估计每个条目占用不少于 255 个字节，该字典只需要大约 12 MB内存。

告知你的面试官你准备写多少代码。

```
class DefaultCategories(Enum):

    HOUSING = 0
    FOOD = 1
    GAS = 2
    SHOPPING = 3
    ...

seller_category_map = {}
seller_category_map['Exxon'] = DefaultCategories.GAS
seller_category_map['Target'] = DefaultCategories.SHOPPING
...
```

对于一开始没有在映射中的卖家，我们可以通过评估用户提供的手动类别来进行众包。在 O(1) 时间内，我们可以用堆来快速查找每个卖家的顶端的手动覆盖。

```
class Categorizer(object):

    def __init__(self, seller_category_map, self.seller_category_crowd_overrides_map):
        self.seller_category_map = seller_category_map
        self.seller_category_crowd_overrides_map = \
            seller_category_crowd_overrides_map

    def categorize(self, transaction):
        if transaction.seller in self.seller_category_map:
            return self.seller_category_map[transaction.seller]
        elif transaction.seller in self.seller_category_crowd_overrides_map:
            self.seller_category_map[transaction.seller] = \
                self.seller_category_crowd_overrides_map[transaction.seller].peek_min(
)
            return self.seller_category_map[transaction.seller]
        return None
```

交易实现：

```
class Transaction(object):

    def __init__(self, created_at, seller, amount):
        self.timestamp = timestamp
        self.seller = seller
        self.amount = amount
```

用例：服务推荐预算

首先，我们可以使用根据收入等级分配每类别金额的通用预算模板。使用这种方法，我们不必存储在约束中标识的 1 亿个预算项目，只需存储用户覆盖的预算项目。如果用户覆盖预算类别，我们可以在 `TABLE budget_overrides` 中存储此覆盖。

```
class Budget(object):

    def __init__(self, income):
        self.income = income
        self.categories_to_budget_map = self.create_budget_template()

    def create_budget_template(self):
        return {
            'DefaultCategories.HOUSING': income * .4,
            'DefaultCategories.FOOD': income * .2
            'DefaultCategories.GAS': income * .1,
            'DefaultCategories.SHOPPING': income * .2
            ...
        }

    def override_category_budget(self, category, amount):
        self.categories_to_budget_map[category] = amount
```

对于 预算服务 而言，我们可以在 `transactions` 表上运行SQL查询以生成 `monthly_spending` 聚合表。由于用户通常每个月有很多交易，所以 `monthly_spending` 表的行数可能会少于总共50亿次交易的行数。

作为替代，我们可以在原始交易文件上运行 **MapReduce** 作业来：

- 分类每个交易
- 按类别生成每月总支出

对交易文件的运行分析可以显著减少数据库的负载。

如果用户更新类别，我们可以调用 预算服务 重新运行分析。

告知你的面试官你准备写多少代码.

日志文件格式样例，以tab分割：

```
user_id   timestamp   seller  amount
```

MapReduce 实现:

```
class SpendingByCategory(MRJob):

    def __init__(self, categorizer):
        self.categorizer = categorizer
        self.current_year_month = calc_current_year_month()
        ...

    def calc_current_year_month(self):
        """返回当前年月"""
        ...

    def extract_year_month(self, timestamp):
        """返回时间戳的年，月部分"""
        ...

    def handle_budget_notifications(self, key, total):
        """如果接近或超出预算，调用通知API"""
        ...

    def mapper(self, _, line):
        """解析每个日志行，提取和转换相关行。

        参数行应为如下形式：

        user_id   timestamp   seller  amount

        使用分类器来将卖家转换成类别，生成如下形式的key-value对：

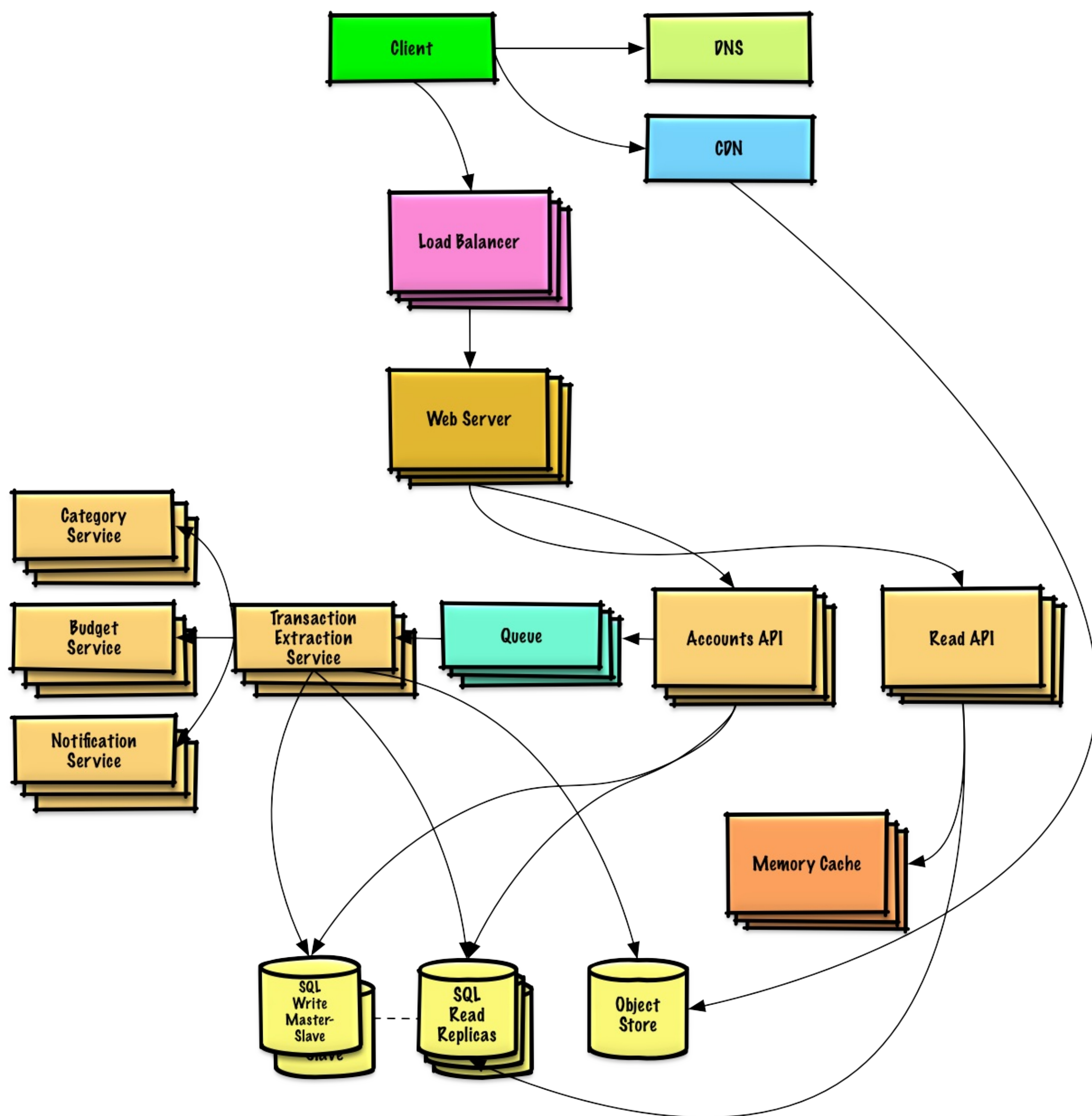
        (user_id, 2016-01, shopping), 25
        (user_id, 2016-01, shopping), 100
        (user_id, 2016-01, gas), 50
        """
        user_id, timestamp, seller, amount = line.split('\t')
        category = self.categorizer.categorize(seller)
        period = self.extract_year_month(timestamp)
        if period == self.current_year_month:
            yield (user_id, period, category), amount

    def reducer(self, key, value):
        """将每个key对应的值求和。

        (user_id, 2016-01, shopping), 125
        (user_id, 2016-01, gas), 50
        """
        total = sum(values)
        yield key, sum(values)
```

第四步：设计扩展

> 根据限制条件，找到并解决瓶颈。



重要提示：不要从最初设计直接跳到最终设计中！

现在你要 1) 基准测试、负载测试。2) 分析、描述性能瓶颈。3) 在解决瓶颈问题的同时，评估替代方案、权衡利弊。4) 重复以上步骤。请阅读「设计一个系统，并将其扩大到为数以百万计的 AWS 用户服务」 来了解如何逐步扩大初始设计。

讨论初始设计可能遇到的瓶颈及相关解决方案是很重要的。例如加上一个配置多台 **Web** 服务器的负载均衡器是否能够解决问题？**CDN**呢？主从复制呢？它们各自的替代方案和需要权衡的利弊又有什么呢？

我们将会介绍一些组件来完成设计，并解决架构扩张问题。内置的负载均衡器将不做讨论以节省篇幅。

为了避免重复讨论，请参考系统设计主题索引相关部分来了解其要点、方案的权衡取舍以及可选的替代方案。

- DNS
- 负载均衡器
- 水平拓展
- 反向代理（web 服务器）
- API 服务（应用层）
- 缓存
- 关系型数据库管理系统 (RDBMS)
- SQL 故障主从切换
- 主从复制
- 一致性模式
- 可用性模式

我们将增加一个额外的用例：用户 访问摘要和交易数据。

用户会话，按类别统计的统计信息，以及最近的事务可以放在 内存缓存（如 Redis 或 Memcached ）中。

- 客户端 发送读请求给 **Web** 服务器
- **Web** 服务器 转发请求到 读 **API** 服务器
    - 静态内容可通过 对象存储 比如缓存在 **CDN** 上的 S3 来服务
- 读 **API** 服务器做如下动作:
    - 检查 内存缓存 的内容
        - 如果URL在 内存缓存中，返回缓存的内容
        - 否则
            - 如果URL在 **SQL** 数据库中，获取该内容
                - 以其内容更新 内存缓存

参考 何时更新缓存 中权衡和替代的内容。以上方法描述了 cache-aside缓存模式.

我们可以使用诸如 Amazon Redshift 或者 Google BigQuery 等数据仓库解决方案，而不是将 `monthly_spending` 聚合表保留在 **SQL** 数据库 中。

我们可能只想在数据库中存储一个月的 `交易` 数据，而将其余数据存储在数据仓库或者 对象存储区 中。对象存储区 （如Amazon S3) 能够舒服地解决每月 250 GB新内容的限制。

为了解决每秒 平均 2000 次读请求数（峰值时更高），受欢迎的内容的流量应由 内存缓存 而不是数据库来处理。 内存缓存 也可用于处理不均匀分布的流量和流量尖峰。 只要副本不陷入重复写入的困境，**SQL** 读副本 应该能够处理高速缓存未命中。

平均 200 次交易写入每秒（峰值时更高）对于单个 **SQL** 写入主-从服务 来说可能是棘手的。我们可能需要考虑其它的 SQL 性能拓展技术：

* 联合
* 分片
* 非规范化
* SQL 调优

我们也可以考虑将一些数据移至 **NoSQL** 数据库。

其它要点

> 是否深入这些额外的主题，取决于你的问题范围和剩下的时间。

NoSQL

* 键-值存储
* 文档类型存储
* 列型存储
* 图数据库
* SQL vs NoSQL

缓存

* 在哪缓存
    * 客户端缓存
    * CDN 缓存
    * Web 服务器缓存
    * 数据库缓存
    * 应用缓存
* 什么需要缓存
    * 数据库查询级别的缓存
    * 对象级别的缓存
* 何时更新缓存
    * 缓存模式
    * 直写模式
    * 回写模式
    * 刷新

异步与微服务

* 消息队列
* 任务队列

- 背压
- 微服务

通信

- 可权衡选择的方案：
  - 与客户端的外部通信 - 使用 REST 作为 HTTP API
  - 服务器内部通信 - RPC
- 服务发现

安全性

请参阅「安全」一章。

延迟数值

请参阅「每个程序员都应该知道的延迟数」。

持续探讨

- 持续进行基准测试并监控你的系统，以解决他们提出的瓶颈问题。
- 架构拓展是一个迭代的过程。

为社交网络设计数据结构

注释：为了避免重复，这篇文章的链接直接关联到 系统设计主题 的相关章节。为一讨论要点、折中方案和可选方案做参考。

第 1 步：用例和约束概要

> 收集需求并调查问题。 通过提问清晰用例和约束。 讨论假设。

如果没有面试官提出明确的问题，我们将自己定义一些用例和约束条件。

用例

我们就处理以下用例审视这一问题

- 用户 寻找某人并显示与被寻人之间的最短路径
- 服务 高可用

约束和假设

状态假设

- 流量分布不均
  - 某些搜索比别的更热门，同时某些搜索仅执行一次
- 图数据不适用单一机器
- 图的边没有权重
- 1 千万用户
- 每个用户平均有 50 个朋友
- 每月 10 亿次朋友搜索

训练使用更传统的系统 - 别用图特有的解决方案例如 GraphQL 或图数据库如 Neo4j。

计算使用

向你的面试官厘清你是否应该做粗略的使用计算

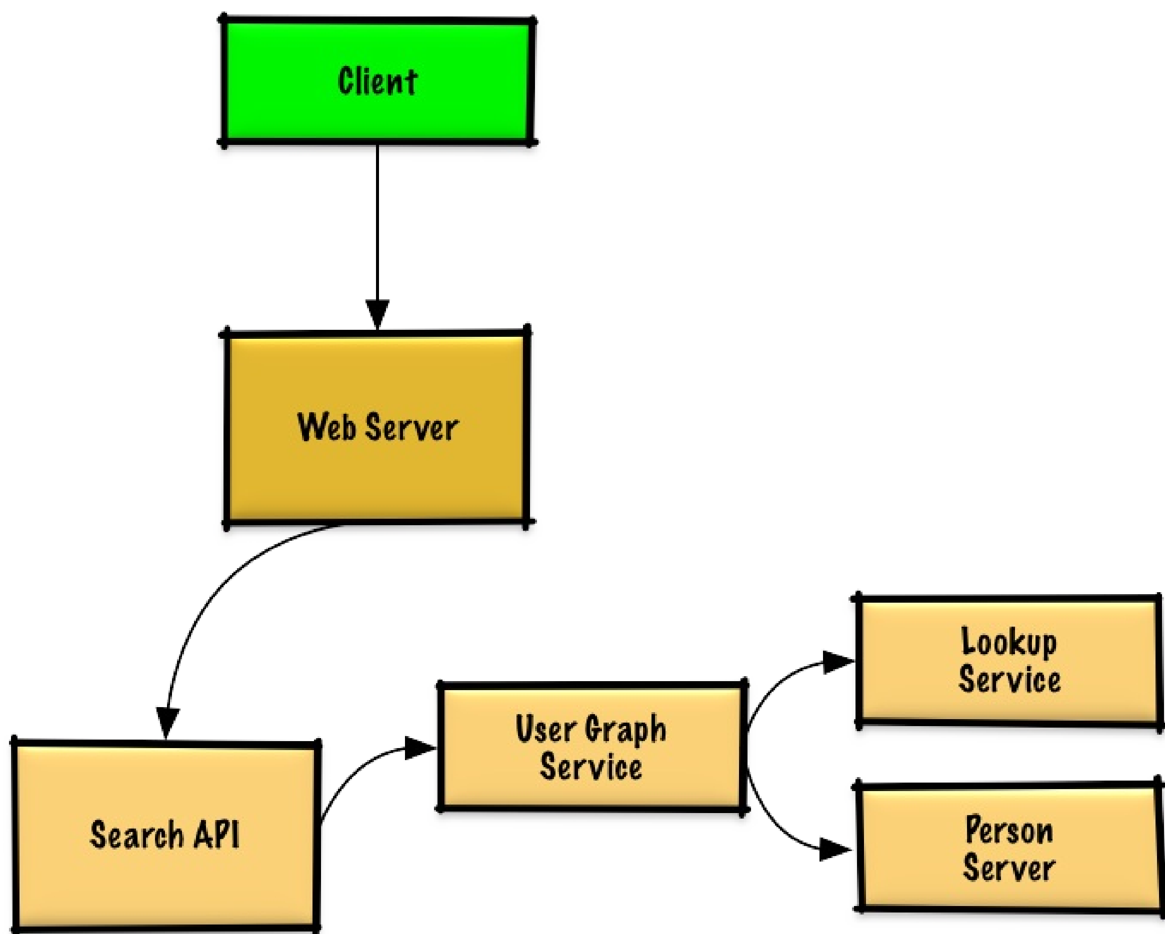
- 50 亿朋友关系
  - 1 亿用户 * 平均每人 50 个朋友
- 每秒 400 次搜索请求

便捷的转换指南：

- 每月 250 万秒
- 每秒 1 个请求 = 每月 250 万次请求
- 每秒 40 个请求 = 每月 1 亿次请求
- 每秒 400 个请求 = 每月 10 亿次请求

第 2 步：创建高级设计方案

> 用所有重要组件概述高水平设计



第 3 步：设计核心组件

> 深入每个核心组件的细节。

用例： 用户搜索某人并查看到被搜人的最短路径

和你的面试官说清你期望的代码量

没有百万用户（点）的和十亿朋友关系（边）的限制，我们能够用一般 BFS 方法解决无权重最短路径任务：

```
class Graph(Graph):

    def shortest_path(self, source, dest):
        if source is None or dest is None:
            return None
        if source is dest:
            return [source.key]
        prev_node_keys = self._shortest_path(source, dest)
        if prev_node_keys is None:
            return None
        else:
            path_ids = [dest.key]
            prev_node_key = prev_node_keys[dest.key]
            while prev_node_key is not None:
                path_ids.append(prev_node_key)
                prev_node_key = prev_node_keys[prev_node_key]
            return path_ids[::-1]

    def _shortest_path(self, source, dest):
        queue = deque()
        queue.append(source)
        prev_node_keys = {source.key: None}
        source.visit_state = State.visited
        while queue:
            node = queue.popleft()
            if node is dest:
                return prev_node_keys
            prev_node = node
            for adj_node in node.adj_nodes.values():
                if adj_node.visit_state == State.unvisited:
                    queue.append(adj_node)
                    prev_node_keys[adj_node.key] = prev_node.key
                    adj_node.visit_state = State.visited
        return None
```

我们不能在同一台机器上满足所有用户，我们需要通过 人员服务器 拆分 用户并且通过 查询服务 访问。

- 客户端 向 服务器 发送请求，服务器 作为 反向代理
- 搜索 **API** 服务器向 用户图服务 转发请求
- 用户图服务 有以下功能：
  - 使用 查询服务 找到当前用户信息存储的 人员服务器
  - 找到适当的 人员服务器 检索当前用户的 `friend_ids` 列表
  - 把当前用户作为 `source` 运行 BFS 搜索算法同时 当前用户的 `friend_ids` 作为每个 `adjacent_node` 的 ids

- 给定 id 获取 `adjacent_node`：
  - 用户图服务 将 再次 和 查询服务 通讯，最后判断出和给定 id 相匹配的存储 `adjacent_node` 的 人员服务器（有待优化）

和你的面试官说清你应该写的代码量

注释：简易版错误处理执行如下。询问你是否需要编写适当的错误处理方法。

查询服务 实现：

```
class LookupService(object):

    def __init__(self):
        self.lookup = self._init_lookup()  # key: person_id, value: person_server

    def _init_lookup(self):
        ...

    def lookup_person_server(self, person_id):
        return self.lookup[person_id]
```

人员服务器 实现：

```
class PersonServer(object):

    def __init__(self):
        self.people = {}  # key: person_id, value: person

    def add_person(self, person):
        ...

    def people(self, ids):
        results = []
        for id in ids:
            if id in self.people:
                results.append(self.people[id])
        return results
```

用户 实现：

```
class Person(object):

    def __init__(self, id, name, friend_ids):
        self.id = id
        self.name = name
        self.friend_ids = friend_ids
```

用户图服务 实现：

```
class UserGraphService(object):

    def __init__(self, lookup_service):
        self.lookup_service = lookup_service

    def person(self, person_id):
        person_server = self.lookup_service.lookup_person_server(person_id)
        return person_server.people([person_id])

    def shortest_path(self, source_key, dest_key):
        if source_key is None or dest_key is None:
            return None
        if source_key is dest_key:
            return [source_key]
        prev_node_keys = self._shortest_path(source_key, dest_key)
        if prev_node_keys is None:
            return None
        else:
            # Iterate through the path_ids backwards, starting at dest_key
            path_ids = [dest_key]
            prev_node_key = prev_node_keys[dest_key]
            while prev_node_key is not None:
                path_ids.append(prev_node_key)
                prev_node_key = prev_node_keys[prev_node_key]
            # Reverse the list since we iterated backwards
            return path_ids[::-1]

    def _shortest_path(self, source_key, dest_key, path):
        # Use the id to get the Person
        source = self.person(source_key)
        # Update our bfs queue
        queue = deque()
        queue.append(source)
        # prev_node_keys keeps track of each hop from
        # the source_key to the dest_key
        prev_node_keys = {source_key: None}
        # We'll use visited_ids to keep track of which nodes we've
        # visited, which can be different from a typical bfs where
        # this can be stored in the node itself
        visited_ids = set()
        visited_ids.add(source.id)
        while queue:
            node = queue.popleft()
            if node.key is dest_key:
                return prev_node_keys
            prev_node = node
            for friend_id in node.friend_ids:
                if friend_id not in visited_ids:
                    friend_node = self.person(friend_id)
                    queue.append(friend_node)
                    prev_node_keys[friend_id] = prev_node.key
                    visited_ids.add(friend_id)
```

```
        return None
```

我们用的是公共的 **REST API**：

```
$ curl https://social.com/api/v1/friend_search?person_id=1234
```

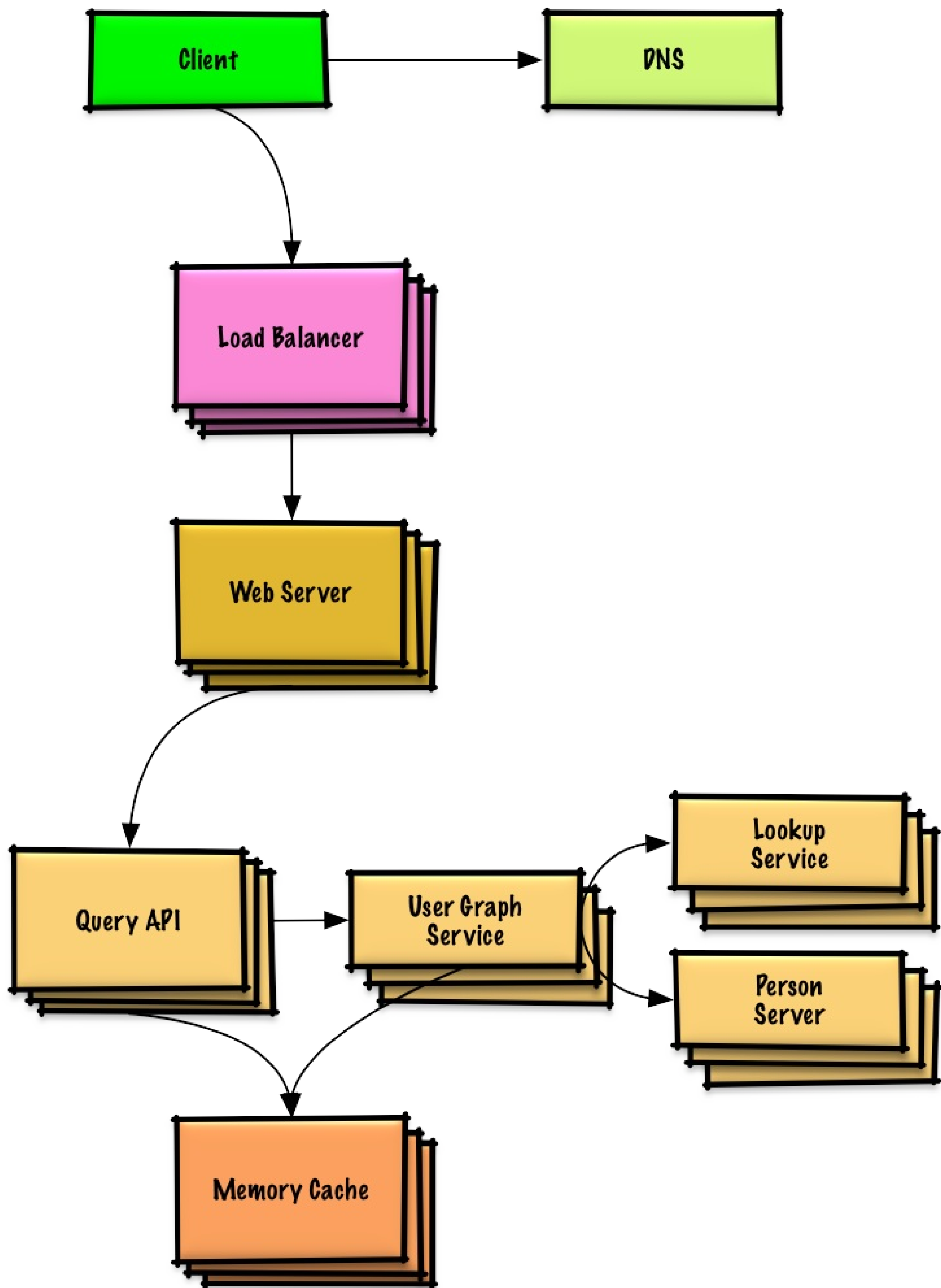
响应：

```
{
    "person_id": "100",
    "name": "foo",
    "link": "https://social.com/foo",
},
{
    "person_id": "53",
    "name": "bar",
    "link": "https://social.com/bar",
},
{
    "person_id": "1234",
    "name": "baz",
    "link": "https://social.com/baz",
},
```

内部通信使用 远端过程调用。

第 4 步：扩展设计

> 在给定约束条件下，定义和确认瓶颈。



重要：别简化从最初设计到最终设计的过程！

你将要做的是：1) 基准/负载 测试， 2) 瓶颈 概述， 3) 当评估可选和折中方案时定位瓶颈，4) 重复。以 在 AWS 上设计支持百万级到千万级用户的系统 为参考迭代地扩展最初设计。

讨论最初设计可能遇到的瓶颈和处理方法十分重要。例如，什么问题可以通过添加多台 **Web 服务器** 作为 负载均衡 解决？**CDN**？主从副本？每个问题都有哪些替代和 折中 方案？

我们即将介绍一些组件来完成设计和解决扩展性问题。内部负载均衡不显示以减少混乱。

避免重复讨论，以下网址链接到 系统设计主题 相关的主流方案、折中方案和替代方案。

- DNS
- 负载均衡
- 横向扩展
- Web 服务器（反向代理）
- API 服务器（应用层）
- 缓存
- 一致性模式
- 可用性模式

解决 平均 每秒 400 次请求的限制（峰值），人员数据可以存在例如 Redis 或 Memcached 这样的 内存 中以减少响应次数和下游流量通信服务。这尤其在用户执行多次连续查询和查询哪些广泛连接的人时十分有用。从内存中读取 1MB 数据大约要 250 微秒，从 SSD 中读取同样大小的数据时间要长 4 倍，从硬盘要长 80 倍。[1]

以下是进一步优化方案：

- 在 内存 中存储完整的或部分的BFS遍历加快后续查找
- 在 **NoSQL** 数据库 中批量离线计算并存储完整的或部分的BFS遍历加快后续查找
- 在同一台 人员服务器 上托管批处理同一批朋友查找减少机器跳转
  - 通过地理位置 拆分 人员服务器 来进一步优化，因为朋友通常住得都比较近
- 同时进行两个 BFS 查找，一个从 source 开始，一个从 destination 开始，然后合并两个路径
- 从有庞大朋友圈的人开始找起，这样更有可能减小当前用户和搜索目标之间的 离散度数
- 在询问用户是否继续查询之前设置基于时间或跳跃数阈值，当在某些案例中搜索耗费时间过长时。
- 使用类似 Neo4j 的 图数据库 或图特定查询语法，例如 GraphQL（如果没有禁止使用 图数据库 的限制的话）

额外的话题

> 根据问题的范围和剩余时间，还需要深入讨论其他问题。

SQL 扩展模式

- 读取副本

- 集合
- 分区
- 反规范化
- SQL 调优

NoSQL

- 键值存储
- 文档存储
- 宽表存储
- 图数据库
- SQL vs NoSQL

缓存

- 缓存到哪里
    - 客户端缓存
    - CDN 缓存
    - Web 服务缓存
    - 数据库缓存
    - 应用缓存
- 缓存什么
    - 数据库请求层缓存
    - 对象层缓存
- 何时更新缓存
    - 预留缓存
    - 完全写入
    - 延迟写 (写回)
    - 事先更新

异步性和微服务

- 消息队列
- 任务队列
- 回退压力
- 微服务

沟通

- 关于折中方案的讨论:
    - 客户端的外部通讯 - 遵循 REST 的 HTTP APIs

    - 内部通讯 - RPC
- 服务探索

安全性

参考 安全章节

延迟数字指标

查阅 每个程序员必懂的延迟数字

正在进行

- 继续基准测试并监控你的系统以解决出现的瓶颈问题
- 扩展是一个迭代的过程

设计一个键-值缓存来存储最近 web 服务查询的结果

注意：这个文档中的链接会直接指向系统设计主题索引中的有关部分，以避免重复的内容。你可以参考链接的相关内容，来了解其总的要点、方案的权衡取舍以及可选的替代方案。

第一步：简述用例与约束条件

> 搜集需求与问题的范围。提出问题来明确用例与约束条件。讨论假设。

我们将在没有面试官明确说明问题的情况下，自己定义一些用例以及限制条件。

用例

我们将把问题限定在仅处理以下用例的范围中

- 用户发送一个搜索请求，命中缓存
- 用户发送一个搜索请求，未命中缓存
- 服务有着高可用性

限制条件与假设

提出假设

- 网络流量不是均匀分布的
    - 经常被查询的内容应该一直存于缓存中
    - 需要确定如何规定缓存过期、缓存刷新规则
- 缓存提供的服务查询速度要快
- 机器间延迟较低
- 缓存有内存限制
    - 需要决定缓存什么、移除什么
    - 需要缓存百万级的查询
- 1000 万用户
- 每个月 100 亿次查询

计算用量

如果你需要进行粗略的用量计算，请向你的面试官说明。

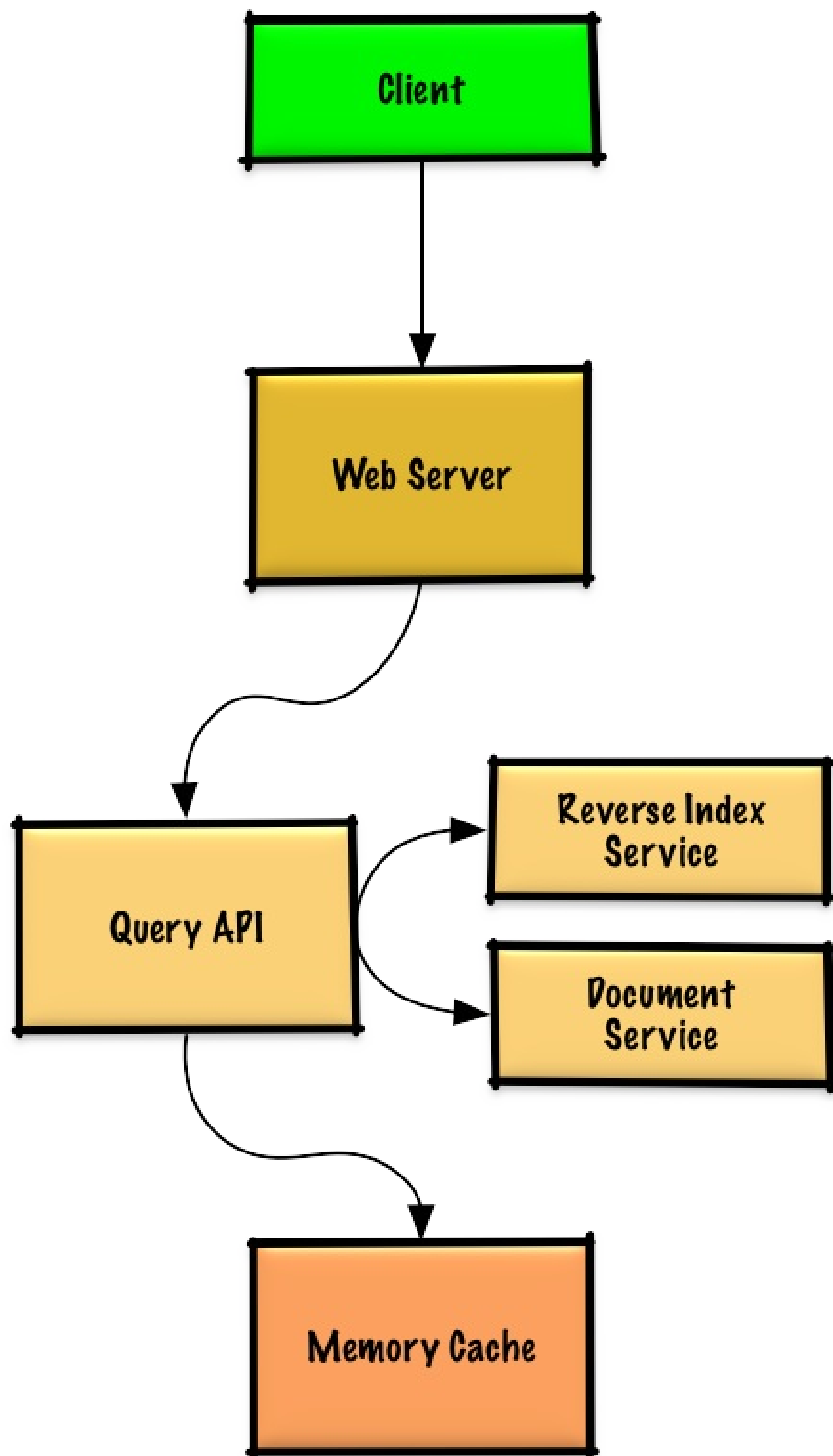
- 缓存存储的是键值对有序表，键为 `query`（查询），值为 `results`（结果）。
    - `query` - 50 字节
    - `title` - 20 字节
    - `snippet` - 200 字节
    - 总计：270 字节
- 假如 100 亿次查询都是不同的，且全部需要存储，那么每个月需要 2.7 TB 的缓存空间
    - 单次查询 270 字节 * 每月查询 100 亿次
    - 假设内存大小有限制，需要决定如何制定缓存过期规则
- 每秒 4,000 次请求

便利换算指南：

- 每个月有 250 万秒
- 每秒一个请求 = 每个月 250 万次请求
- 每秒 40 个请求 = 每个月 1 亿次请求
- 每秒 400 个请求 = 每个月 10 亿次请求

第二步：概要设计

> 列出所有重要组件以规划概要设计。



第三步：设计核心组件

> 深入每个核心组件的细节。

用例：用户发送了一次请求，命中了缓存

常用的查询可以由例如 Redis 或者 Memcached 之类的内存缓存提供支持，以减少数据读取延迟，并且避免反向索引服务以及文档服务的过载。从内存读取 1 MB 连续数据大约要花 250 微秒，而从 SSD 读取同样大小的数据要花费 4 倍的时间，从机械硬盘读取需要花费 80 倍以上的时间。[1]

由于缓存容量有限，我们将使用 LRU（近期最少使用算法）来控制缓存的过期。

- 客户端向运行反向代理的 **Web** 服务器发送一个请求
- 这个 **Web** 服务器将请求转发给查询 **API** 服务
- 查询 **API** 服务将会做这些事情：
  - 分析查询
    - 移除多余的内容
    - 将文本分割成词组
    - 修正拼写错误
    - 规范化字母的大小写
    - 将查询转换为布尔运算
  - 检测内存缓存是否有匹配查询的内容
    - 如果命中内存缓存，内存缓存将会做以下事情：
      - 将缓存入口的位置指向 LRU 链表的头部
      - 返回缓存内容
    - 否则，查询 **API** 将会做以下事情：
      - 使用反向索引服务来查找匹配查询的文档
        - 反向索引服务对匹配到的结果进行排名，然后返回最符合的结果
      - 使用文档服务返回文章标题与片段
      - 更新内存缓存，存入内容，将内存缓存入口位置指向 LRU 链表的头部

缓存的实现

缓存可以使用双向链表实现：新元素将会在头结点加入，过期的元素将会在尾节点被删除。我们使用哈希表以便能够快速查找每个链表节点。

向你的面试官告知你准备写多少代码。

实现查询 **API** 服务：

```
class QueryApi(object):

    def __init__(self, memory_cache, reverse_index_service):
        self.memory_cache = memory_cache
        self.reverse_index_service = reverse_index_service

    def parse_query(self, query):
        """移除多余内容，将文本分割成词组，修复拼写错误，
        规范化字母大小写，转换布尔运算。
        """
        ...

    def process_query(self, query):
        query = self.parse_query(query)
        results = self.memory_cache.get(query)
        if results is None:
            results = self.reverse_index_service.process_search(query)
            self.memory_cache.set(query, results)
        return results
```

实现节点：

```
class Node(object):

    def __init__(self, query, results):
        self.query = query
        self.results = results
```

实现链表：

```
class LinkedList(object):

    def __init__(self):
        self.head = None
        self.tail = None

    def move_to_front(self, node):
        ...

    def append_to_front(self, node):
        ...

    def remove_from_tail(self):
        ...
```

实现缓存：

```
class Cache(object):

    def __init__(self, MAX_SIZE):
        self.MAX_SIZE = MAX_SIZE
        self.size = 0
        self.lookup = {}  # key: query, value: node
        self.linked_list = LinkedList()

    def get(self, query)
        """从缓存取得存储的内容

        将入口节点位置更新为 LRU 链表的头部。
        """
        node = self.lookup[query]
        if node is None:
            return None
        self.linked_list.move_to_front(node)
        return node.results

    def set(self, results, query):
        """将所给查询键的结果存在缓存中。

        当更新缓存记录的时候，将它的位置指向 LRU 链表的头部。
        如果这个记录是新的记录，并且缓存空间已满，应该在加入新记录前
        删除最老的记录。
        """
        node = self.lookup[query]
        if node is not None:
            # 键存在于缓存中，更新它对应的值
            node.results = results
            self.linked_list.move_to_front(node)
        else:
            # 键不存在于缓存中
            if self.size == self.MAX_SIZE:
                # 在链表中查找并删除最老的记录
                self.lookup.pop(self.linked_list.tail.query, None)
                self.linked_list.remove_from_tail()
            else:
                self.size += 1
            # 添加新的键值对
            new_node = Node(query, results)
            self.linked_list.append_to_front(new_node)
            self.lookup[query] = new_node
```

何时更新缓存

缓存将会在以下几种情况更新：

- 页面内容发生变化
- 页面被移除或者加入了新页面

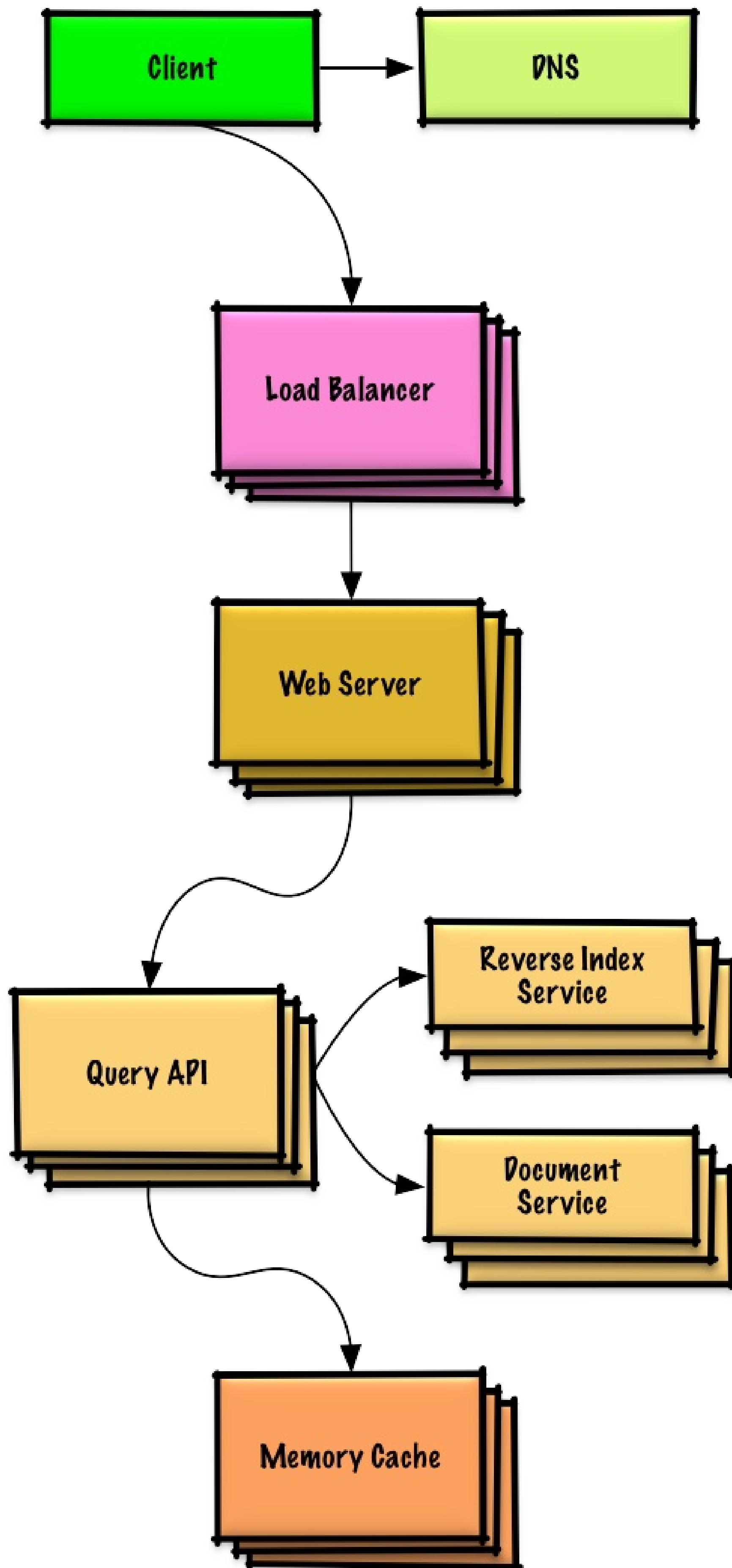
- 页面的权值发生变动

解决这些问题的最直接的方法，就是为缓存记录设置一个它在被更新前能留在缓存中的最长时间，这个时间简称为存活时间（TTL）。

参考「何时更新缓存」来了解其权衡取舍及替代方案。以上方法在缓存模式一章中详细地进行了描述。

第四步：架构扩展

> 根据限制条件，找到并解决瓶颈。


Client
DNS
Load Balancer
Web Server
Query API
Reverse Index Service
Document Service
Memory Cache

重要提示：不要从最初设计直接跳到最终设计中！

现在你要 1) 基准测试、负载测试。2) 分析、描述性能瓶颈。3) 在解决瓶颈问题的同时，评估替代方案、权衡利弊。4) 重复以上步骤。请阅读「设计一个系统，并将其扩大到为数以百万计的 AWS 用户服务」 来了解如何逐步扩大初始设计。

讨论初始设计可能遇到的瓶颈及相关解决方案是很重要的。例如加上一个配置多台 **Web** 服务器的负载均衡器是否能够解决问题？**CDN**呢？主从复制呢？它们各自的替代方案和需要权衡的利弊又有什么呢？

我们将会介绍一些组件来完成设计，并解决架构扩张问题。内置的负载均衡器将不做讨论以节省篇幅。

为了避免重复讨论，请参考系统设计主题索引相关部分来了解其要点、方案的权衡取舍以及可选的替代方案。

- DNS
- 负载均衡器
- 水平拓展
- 反向代理（web 服务器）
- API 服务（应用层）
- 缓存
- 一致性模式
- 可用性模式

将内存缓存扩大到多台机器

为了解决庞大的请求负载以及巨大的内存需求，我们将要对架构进行水平拓展。如何在我们的内存缓存集群中存储数据呢？我们有以下三个主要可选方案：

- 缓存集群中的每一台机器都有自己的缓存 - 简单，但是它会降低缓存命中率。
- 缓存集群中的每一台机器都有缓存的拷贝 - 简单，但是它的内存使用效率太低了。
- 对缓存进行分片，分别部署在缓存集群中的所有机器中 - 更加复杂，但是它是最佳的选择。我们可以使用哈希，用查询语句 `machine = hash(query)` 来确定哪台机器有需要缓存。当然我们也可以使用一致性哈希。

其它要点

> 是否深入这些额外的主题，取决于你的问题范围和剩下的时间。

SQL 缩放模式

- 读取复制
- 联合
- 分片
- 非规范化
- SQL 调优

NoSQL

- 键-值存储
- 文档类型存储
- 列型存储
- 图数据库
- SQL vs NoSQL

缓存

- 在哪缓存
    - 客户端缓存
    - CDN 缓存
    - Web 服务器缓存
    - 数据库缓存
    - 应用缓存
- 什么需要缓存
    - 数据库查询级别的缓存
    - 对象级别的缓存
- 何时更新缓存
    - 缓存模式
    - 直写模式
    - 回写模式
    - 刷新

异步与微服务

- 消息队列
- 任务队列
- 背压
- 微服务

通信

- 可权衡选择的方案：

    - 与客户端的外部通信 - 使用 REST 作为 HTTP API
    - 服务器内部通信 - RPC
- 服务发现

安全性

请参阅「安全」一章。

延迟数值

请参阅「每个程序员都应该知道的延迟数」。

持续探讨

- 持续进行基准测试并监控你的系统，以解决他们提出的瓶颈问题。
- 架构拓展是一个迭代的过程。

为 Amazon 设计分类售卖排行

注意：这个文档中的链接会直接指向系统设计主题索引中的有关部分，以避免重复的内容。你可以参考链接的相关内容，来了解其总的要点、方案的权衡取舍以及可选的替代方案。

第一步：简述用例与约束条件

> 搜集需求与问题的范围。 提出问题来明确用例与约束条件。 讨论假设。

我们将在没有面试官明确说明问题的情况下，自己定义一些用例以及限制条件。

用例

我们将把问题限定在仅处理以下用例的范围中

- 服务根据分类计算过去一周中最受欢迎的商品
- 用户通过分类浏览过去一周中最受欢迎的商品
- 服务有着高可用性

不在用例范围内的有

- 一般的电商网站
  - 只为售卖排行榜设计组件

限制条件与假设

提出假设

- 网络流量不是均匀分布的
- 一个商品可能存在于多个分类中
- 商品不能够更改分类
- 不会存在如 `foo/bar/baz` 之类的子分类
- 每小时更新一次结果
  - 受欢迎的商品越多，就需要更频繁地更新
- 1000 万个商品
- 1000 个分类
- 每个月 10 亿次交易
- 每个月 1000 亿次读取请求

* 100:1 的读写比例

计算用量

如果你需要进行粗略的用量计算，请向你的面试官说明。

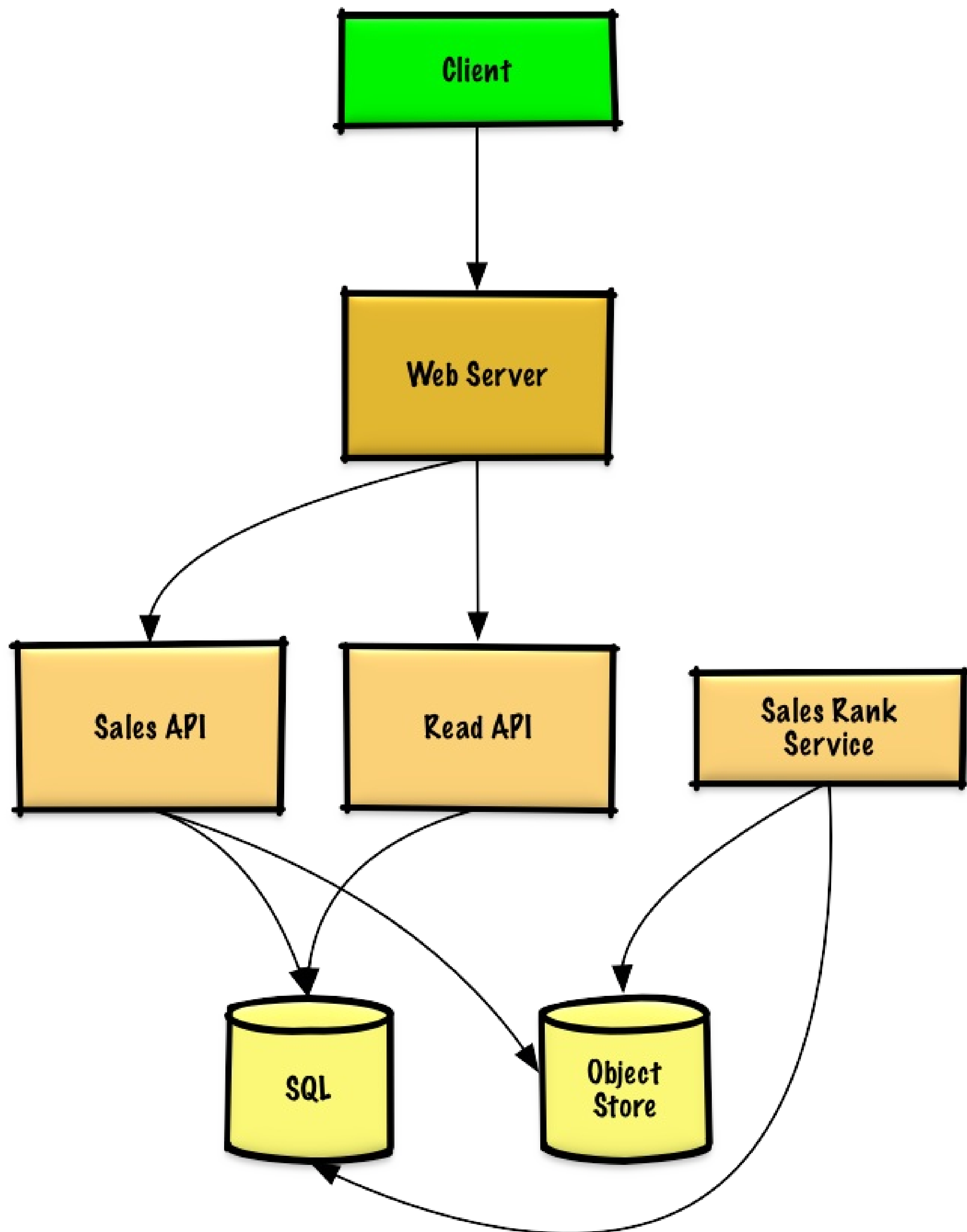
* 每笔交易的用量：
    * `created_at` - 5 字节
    * `product_id` - 8 字节
    * `category_id` - 4 字节
    * `seller_id` - 8 字节
    * `buyer_id` - 8 字节
    * `quantity` - 4 字节
    * `total_price` - 5 字节
    * 总计：大约 40 字节
* 每个月的交易内容会产生 40 GB 的记录
    * 每次交易 40 字节 * 每个月 10 亿次交易
    * 3年内产生了 1.44 TB 的新交易内容记录
    * 假定大多数的交易都是新交易而不是更改以前进行完的交易
* 平均每秒 400 次交易次数
* 平均每秒 40,000 次读取请求

便利换算指南：

* 每个月有 250 万秒
* 每秒一个请求 = 每个月 250 万次请求
* 每秒 40 个请求 = 每个月 1 亿次请求
* 每秒 400 个请求 = 每个月 10 亿次请求

第二步：概要设计

> 列出所有重要组件以规划概要设计。



第三步：设计核心组件

> 深入每个核心组件的细节。

用例：服务需要根据分类计算上周最受欢迎的商品

我们可以在现成的对象存储系统（例如 Amazon S3 服务）中存储 售卖 **API** 服务产生的日志文本， 因此不需要我们自己搭建分布式文件系统了。

向你的面试官告知你准备写多少代码。

假设下面是一个用 tab 分割的简易的日志记录：

```
timestamp   product_id  category_id    qty     total_price   seller_id    buyer_id
t1          product1    category1      2       20.00         1            1
t2          product1    category2      2       20.00         2            2
t2          product1    category2      1       10.00         2            3
t3          product2    category1      3        7.00         3            4
t4          product3    category2      7        2.00         4            5
t5          product4    category1      1        5.00         5            6
...
```

售卖排行服务 需要用到 **MapReduce**，并使用 售卖 **API** 服务进行日志记录，同时将结果写入 **SQL** 数据库中的总表 `sales_rank` 中。我们也可以讨论一下究竟是用 SQL 还是用 NoSQL。

我们需要通过以下步骤使用 **MapReduce**：

- 第 **1** 步 - 将数据转换为 `(category, product_id), sum(quantity)` 的形式
- 第 **2** 步 - 执行分布式排序

```
class SalesRanker(MRJob):

    def within_past_week(self, timestamp):
        """如果时间戳属于过去的一周则返回 True，
        否则返回 False。"""
        ...

    def mapper(self, _ line):
        """解析日志的每一行，提取并转换相关行，

        将键值对设定为如下形式：

        (category1, product1), 2
        (category2, product1), 2
        (category2, product1), 1
        (category1, product2), 3
        (category2, product3), 7
        (category1, product4), 1
        """
        timestamp, product_id, category_id, quantity, total_price, seller_id, \
            buyer_id = line.split('\t')
        if self.within_past_week(timestamp):
            yield (category_id, product_id), quantity

    def reducer(self, key, value):
        """将每个 key 的值加起来。
```

```
        (category1, product1), 2
        (category2, product1), 3
        (category1, product2), 3
        (category2, product3), 7
        (category1, product4), 1
        """
        yield key, sum(values)

    def mapper_sort(self, key, value):
        """构造 key 以确保正确的排序。

        将键值对转换成如下形式：

        (category1, 2), product1
        (category2, 3), product1
        (category1, 3), product2
        (category2, 7), product3
        (category1, 1), product4

        MapReduce 的随机排序步骤会将键
        值的排序打乱，变成下面这样：

        (category1, 1), product4
        (category1, 2), product1
        (category1, 3), product2
        (category2, 3), product1
        (category2, 7), product3
        """
        category_id, product_id = key
        quantity = value
        yield (category_id, quantity), product_id

    def reducer_identity(self, key, value):
        yield key, value

    def steps(self):
        """ 此处为 map reduce 步骤"""
        return [
            self.mr(mapper=self.mapper,
                    reducer=self.reducer),
            self.mr(mapper=self.mapper_sort,
                    reducer=self.reducer_identity),
        ]
```

得到的结果将会是如下的排序列，我们将其插入 `sales_rank` 表中：

```
(category1, 1), product4
(category1, 2), product1
(category1, 3), product2
(category2, 3), product1
(category2, 7), product3
```

`sales_rank` 表的数据结构如下：

```
id int NOT NULL AUTO_INCREMENT
category_id int NOT NULL
total_sold int NOT NULL
product_id int NOT NULL
PRIMARY KEY(id)
FOREIGN KEY(category_id) REFERENCES Categories(id)
FOREIGN KEY(product_id) REFERENCES Products(id)
```

我们会以 `id` 、`category_id` 与 `product_id` 创建一个索引以加快查询速度（只需要使用读取日志的时间，不再需要每次都扫描整个数据表）并让数据常驻内存。从内存读取 1 MB 连续数据大约要花 250 微秒，而从 SSD 读取同样大小的数据要花费 4 倍的时间，从机械硬盘读取需要花费 80 倍以上的时间。[1]

用例：用户需要根据分类浏览上周中最受欢迎的商品

- 客户端向运行反向代理的 **Web** 服务器发送一个请求
- 这个 **Web** 服务器将请求转发给查询 **API** 服务
- The 查询 **API** 服务将从 **SQL** 数据库的 `sales_rank` 表中读取数据

我们可以调用一个公共的 REST API：

```
$ curl https://amazon.com/api/v1/popular?category_id=1234
```

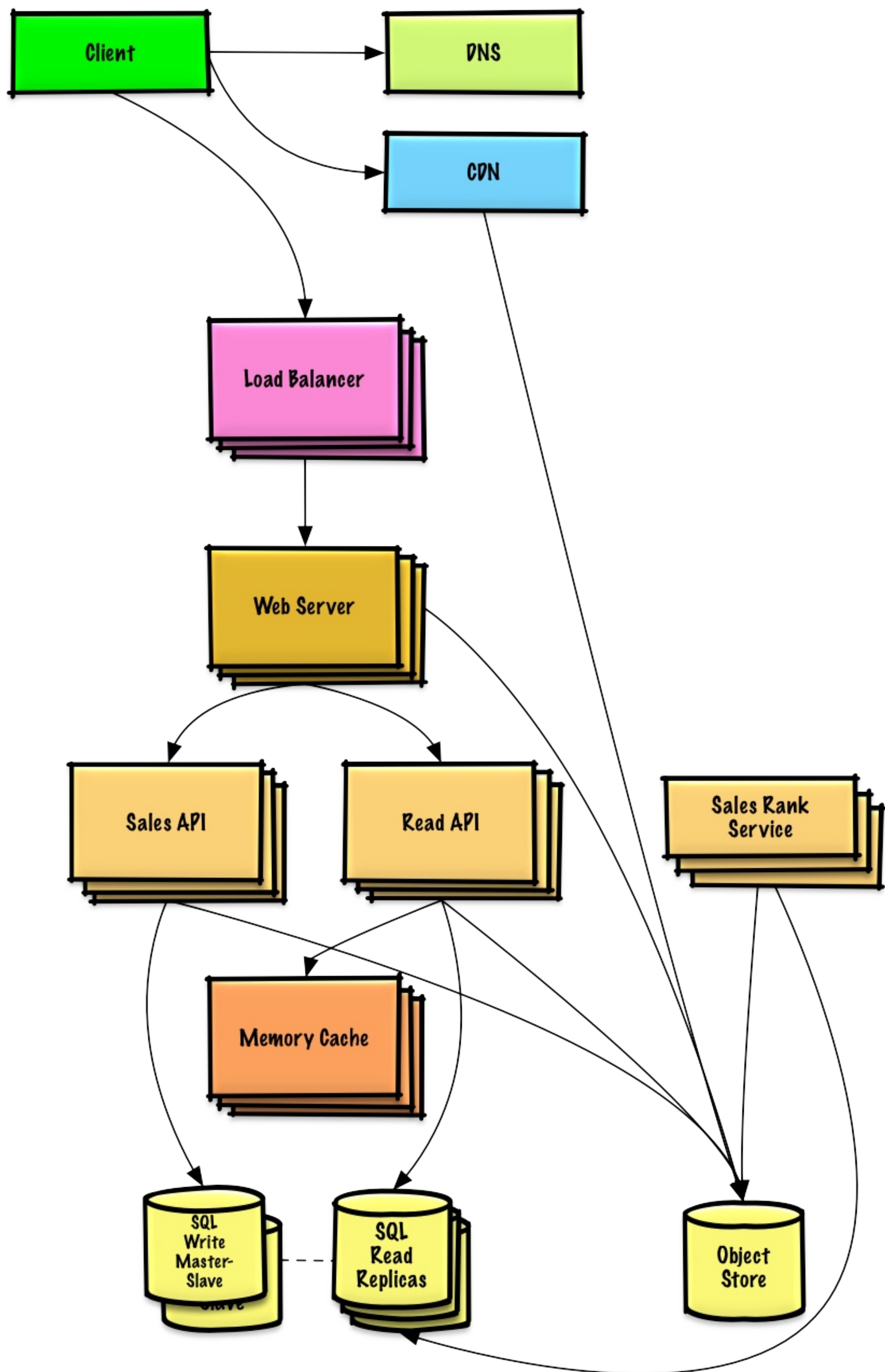
返回：

```
{
    "id": "100",
    "category_id": "1234",
    "total_sold": "100000",
    "product_id": "50",
},
{
    "id": "53",
    "category_id": "1234",
    "total_sold": "90000",
    "product_id": "200",
},
{
    "id": "75",
    "category_id": "1234",
    "total_sold": "80000",
    "product_id": "3",
},
```

而对于服务器内部的通信，我们可以使用 RPC。

第四步：架构扩展

> 根据限制条件，找到并解决瓶颈。



重要提示：不要从最初设计直接跳到最终设计中！

现在你要 1) 基准测试、负载测试。2) 分析、描述性能瓶颈。3) 在解决瓶颈问题的同时，评估替代方案、权衡利弊。4) 重复以上步骤。请阅读「设计一个系统，并将其扩大到为数以百万计的 AWS 用户服务」 来了解如何逐步扩大初始设计。

讨论初始设计可能遇到的瓶颈及相关解决方案是很重要的。例如加上一个配置多台 **Web** 服务器的负载均衡器是否能够解决问题？**CDN**呢？主从复制呢？它们各自的替代方案和需要权衡的利弊又有什么呢？

我们将会介绍一些组件来完成设计，并解决架构扩张问题。内置的负载均衡器将不做讨论以节省篇幅。

为了避免重复讨论，请参考系统设计主题索引相关部分来了解其要点、方案的权衡取舍以及可选的替代方案。

- DNS
- 负载均衡器
- 水平拓展
- 反向代理（web 服务器）
- API 服务（应用层）
- 缓存
- 关系型数据库管理系统 (RDBMS)
- SQL 故障主从切换
- 主从复制
- 一致性模式
- 可用性模式

分析数据库 可以用现成的数据仓储系统，例如使用 Amazon Redshift 或者 Google BigQuery 的解决方案。

当使用数据仓储技术或者对象存储系统时，我们只想在数据库中存储有限时间段的数据。Amazon S3 的对象存储系统可以很方便地设置每个月限制只允许新增 40 GB 的存储内容。

平均每秒 40,000 次的读取请求（峰值将会更高）, 可以通过扩展 内存缓存 来处理热点内容的读取流量，这对于处理不均匀分布的流量和流量峰值也很有用。由于读取量非常大，**SQL Read** 副本 可能会遇到处理缓存未命中的问题，我们可能需要使用额外的 SQL 扩展模式。

平均每秒 400 次写操作（峰值将会更高）可能对于单个 **SQL** 写主**-**从 模式来说比较很困难，因此同时还需要更多的扩展技术

SQL 缩放模式包括：

- 联合
- 分片
- 非规范化
- SQL 调优

我们也可以考虑将一些数据移至 **NoSQL** 数据库。

其它要点

> 是否深入这些额外的主题，取决于你的问题范围和剩下的时间。

NoSQL

- 键-值存储
- 文档类型存储
- 列型存储
- 图数据库
- SQL vs NoSQL

缓存

- 在哪缓存
  - 客户端缓存
  - CDN 缓存
  - Web 服务器缓存
  - 数据库缓存
  - 应用缓存
- 什么需要缓存
  - 数据库查询级别的缓存
  - 对象级别的缓存
- 何时更新缓存
  - 缓存模式
  - 直写模式
  - 回写模式
  - 刷新

异步与微服务

- 消息队列
- 任务队列
- 背压
- 微服务

通信

- 可权衡选择的方案：
  - 与客户端的外部通信 - 使用 REST 作为 HTTP API
  - 服务器内部通信 - RPC
- 服务发现

安全性

请参阅「安全」一章。

延迟数值

请参阅「每个程序员都应该知道的延迟数」。

持续探讨

- 持续进行基准测试并监控你的系统，以解决他们提出的瓶颈问题。
- 架构拓展是一个迭代的过程。

在 AWS 上设计支持百万级到千万级用户的系统

注释：为了避免重复，这篇文章的链接直接关联到 系统设计主题 的相关章节。为一讨论要点、折中方案和可选方案做参考。

第 1 步：用例和约束概要

> 收集需求并调查问题。 通过提问清晰用例和约束。 讨论假设。

如果没有面试官提出明确的问题，我们将自己定义一些用例和约束条件。

用例

解决这个问题是一个循序渐进的过程：1) 基准/负载 测试， 2) 瓶颈 概述， 3) 当评估可选和折中方案时定位瓶颈，4) 重复，这是向可扩展的设计发展基础设计的好模式。

除非你有 AWS 的背景或者正在申请需要 AWS 知识的相关职位，否则不要求了解 AWS 的相关细节。并且，这个练习中讨论的许多原则可以更广泛地应用于AWS生态系统之外。

我们就处理以下用例讨论这一问题

- 用户 进行读或写请求
  - 服务 进行处理，存储用户数据，然后返回结果
- 服务 需要从支持小规模用户开始到百万用户
  - 在我们演化架构来处理大量的用户和请求时，讨论一般的扩展模式
- 服务 高可用

约束和假设

状态假设

- 流量不均匀分布
- 需要关系数据
- 从一个用户扩展到千万用户
  - 表示用户量的增长
    - 用户量+
    - 用户量++

        - 用户量+++
        - ...
    - 1000 万用户
    - 每月 10 亿次写入
    - 每月 1000 亿次读出
    - 100:1 读写比率
    - 每次写入 1 KB 内容

计算使用

向你的面试官厘清你是否应该做粗略的使用计算

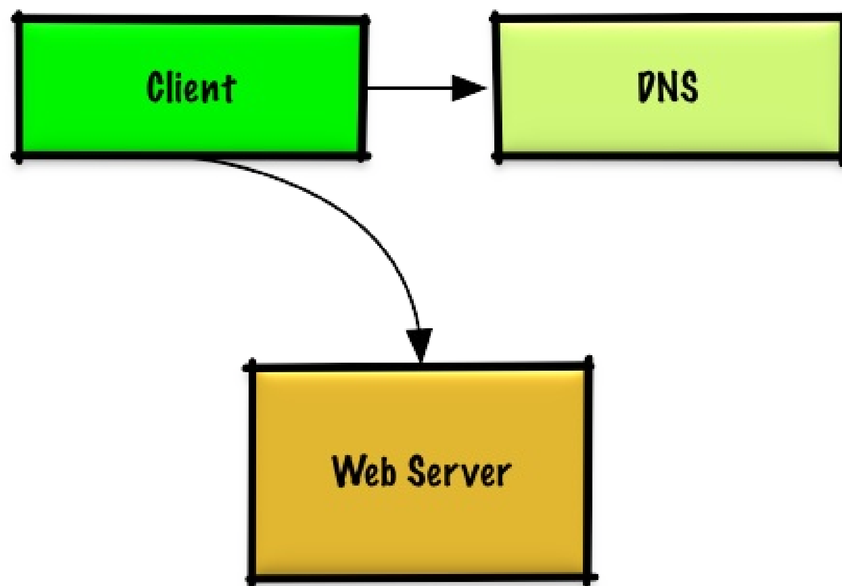
- 1 TB 新内容 / 月
    - 1 KB 每次写入 * 10 亿 写入 / 月
    - 36 TB 新内容 / 3 年
    - 假设大多数写入都是新内容而不是更新已有内容
- 平均每秒 400 次写入
- 平均每秒 40,000 次读取

便捷的转换指南：

- 250 万秒 / 月
- 1 次请求 / 秒 = 250 万次请求 / 月
- 40 次请求 / 秒 = 1 亿次请求 / 月
- 400 次请求 / 秒 = 10 亿请求 / 月

第 2 步：创建高级设计方案

> 用所有重要组件概述高水平设计



第 3 步：设计核心组件

> 深入每个核心组件的细节。

用例：用户进行读写请求

目标

- 只有 1-2 个用户时，你只需要基础配置
    - 为简单起见，只需要一台服务器
    - 必要时进行纵向扩展
    - 监控以确定瓶颈

以单台服务器开始

- **Web** 服务器在 EC2 上
    - 存储用户数据
    - MySQL 数据库

运用 纵向扩展：

- 选择一台更大容量的服务器
- 密切关注指标，确定如何扩大规模
    - 使用基本监控来确定瓶颈:CPU、内存、IO、网络等
    - CloudWatch, top, nagios, statsd, graphite等

* 纵向扩展的代价将变得更昂贵
* 无冗余/容错

折中方案, 可选方案, 和其他细节:

* 纵向扩展 的可选方案是 横向扩展

自 SQL 开始，但认真考虑 NoSQL

约束条件假设需要关系型数据。我们可以开始时在单台服务器上使用 **MySQL** 数据库。

折中方案, 可选方案, 和其他细节:

* 查阅 关系型数据库管理系统 (RDBMS) 章节
* 讨论使用 SQL 或 NoSQL 的原因

分配公共静态 IP

* 弹性 IP 提供了一个公共端点，不会在重启时改变 IP。
* 故障转移时只需要把域名指向新 IP。

使用 DNS 服务

添加 **DNS** 服务，比如 Route 53（Amazon Route 53 - 译者注），将域映射到实例的公共 IP 中。

折中方案, 可选方案, 和其他细节:

* 查阅 域名系统 章节

安全的 Web 服务器

* 只开放必要的端口
    * 允许 Web 服务器响应来自以下端口的请求
        * HTTP 80
        * HTTPS 443
        * SSH IP 白名单 22
    * 防止 Web 服务器启动外链

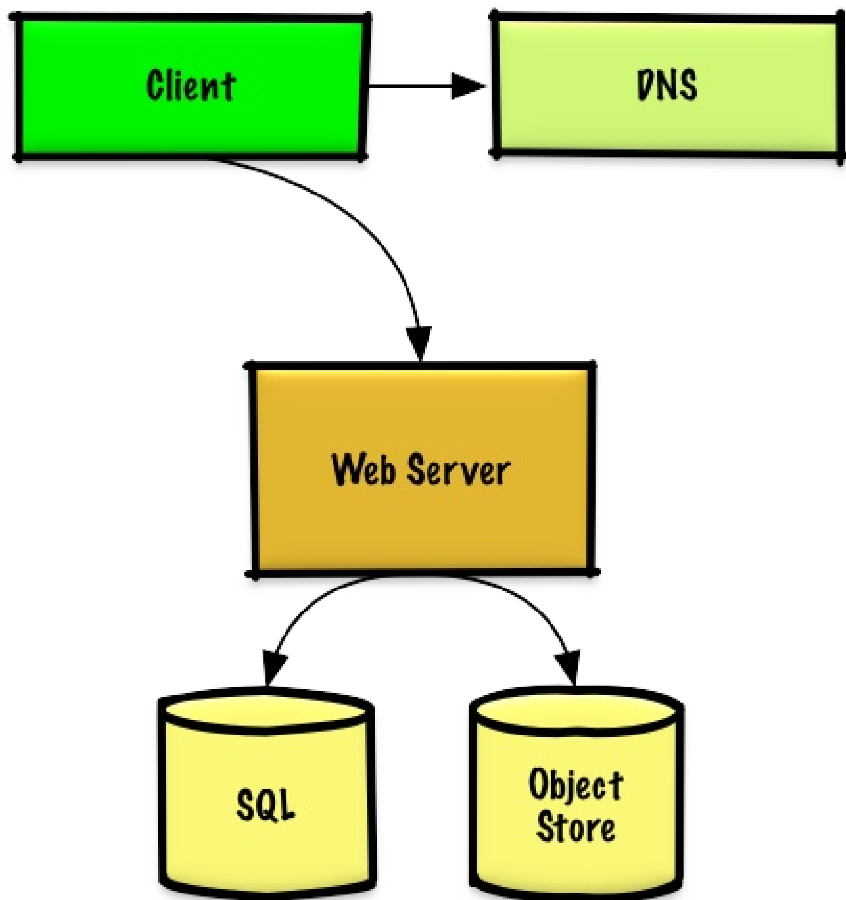
折中方案, 可选方案, 和其他细节:

* 查阅 安全 章节

第 4 步：扩展设计

> 在给定约束条件下，定义和确认瓶颈。

用户+



假设

我们的用户数量开始上升，并且单台服务器的负载上升。基准/负载测试 和 分析 指出 **MySQL** 数据库 占用越来越多的内存和 CPU 资源，同时用户数据将填满硬盘空间。

目前，我们尚能在纵向扩展时解决这些问题。不幸的是，解决这些问题的代价变得相当昂贵，并且原来的系统并不能允许在 **MySQL** 数据库 和 **Web** 服务器 的基础上进行独立扩展。

目标

- 减轻单台服务器负载并且允许独立扩展
    - 在 对象存储 中单独存储静态内容
    - 将 **MySQL** 数据库 迁移到单独的服务器上
- 缺点
    - 这些变化会增加复杂性，并要求对 **Web**服务器 进行更改，以指向 对象存储 和

    MySQL 数据库
    - 必须采取额外的安全措施来确保新组件的安全
    - AWS 的成本也会增加，但应该与自身管理类似系统的成本做比较

独立保存静态内容

- 考虑使用像 S3 这样可管理的 对象存储 服务来存储静态内容
    - 高扩展性和可靠性
    - 服务器端加密
- 迁移静态内容到 S3
    - 用户文件
    - JS
    - CSS
    - 图片
    - 视频

迁移 **MySQL** 数据库到独立机器上

- 考虑使用类似 RDS 的服务来管理 **MySQL** 数据库
    - 简单的管理，扩展
    - 多个可用区域
    - 空闲时加密

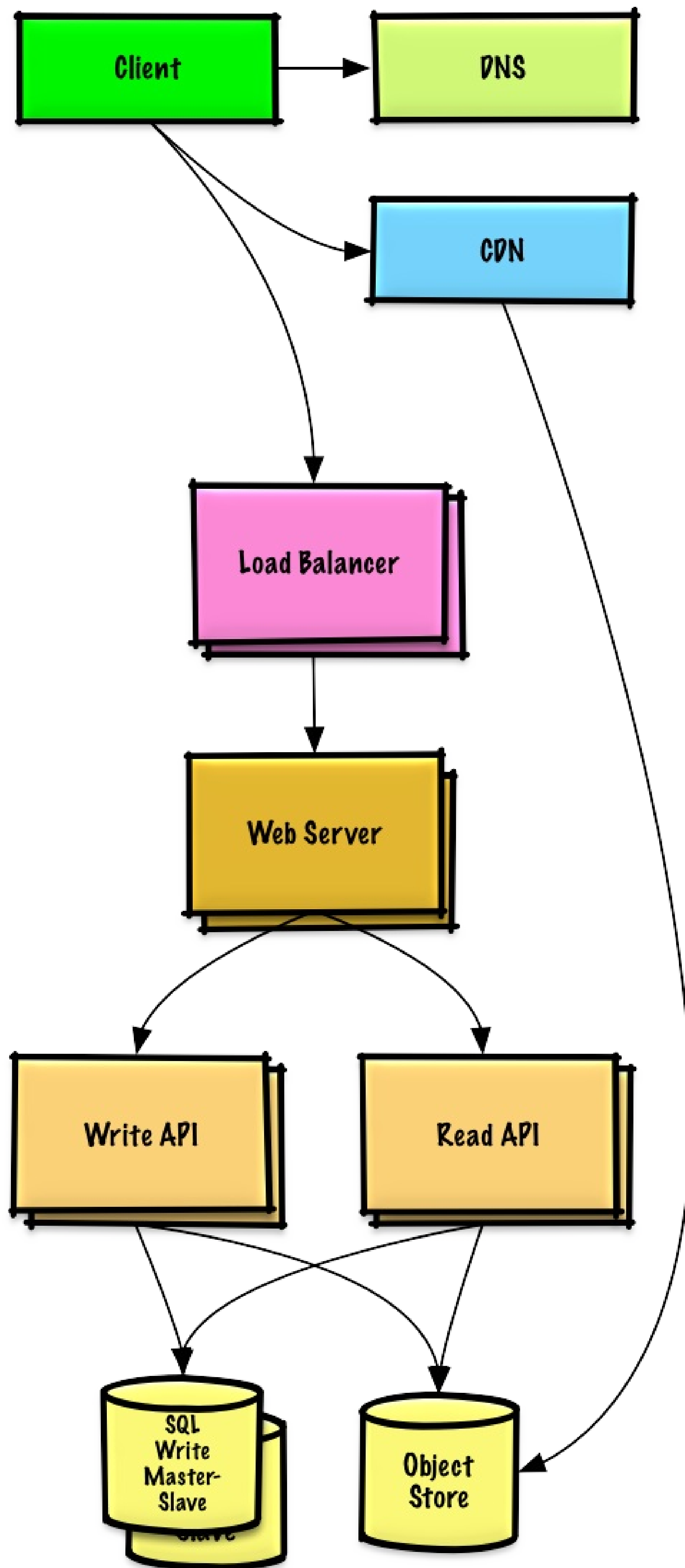
系统安全

- 在传输和空闲时对数据进行加密
- 使用虚拟私有云
    - 为单个 **Web** 服务器 创建一个公共子网，这样就可以发送和接收来自 internet 的流量
    - 为其他内容创建一个私有子网，禁止外部访问
    - 在每个组件上只为白名单 IP 打开端口
- 这些相同的模式应当在新的组件的实现中实践

折中方案, 可选方案, 和其他细节:

- 查阅 安全 章节

用户 **+++**


Client
DNS
CDN
Load Balancer
Web Server
Write API
Read API
SQL
Write
Master-
Slave
Object
Store

假设

我们的 基准/负载测试 和 性能测试 显示，在高峰时段，我们的单一 **Web**服务器 存在瓶颈，导致响应缓慢，在某些情况下还会宕机。随着服务的成熟，我们也希望朝着更高的可用性和冗余发展。

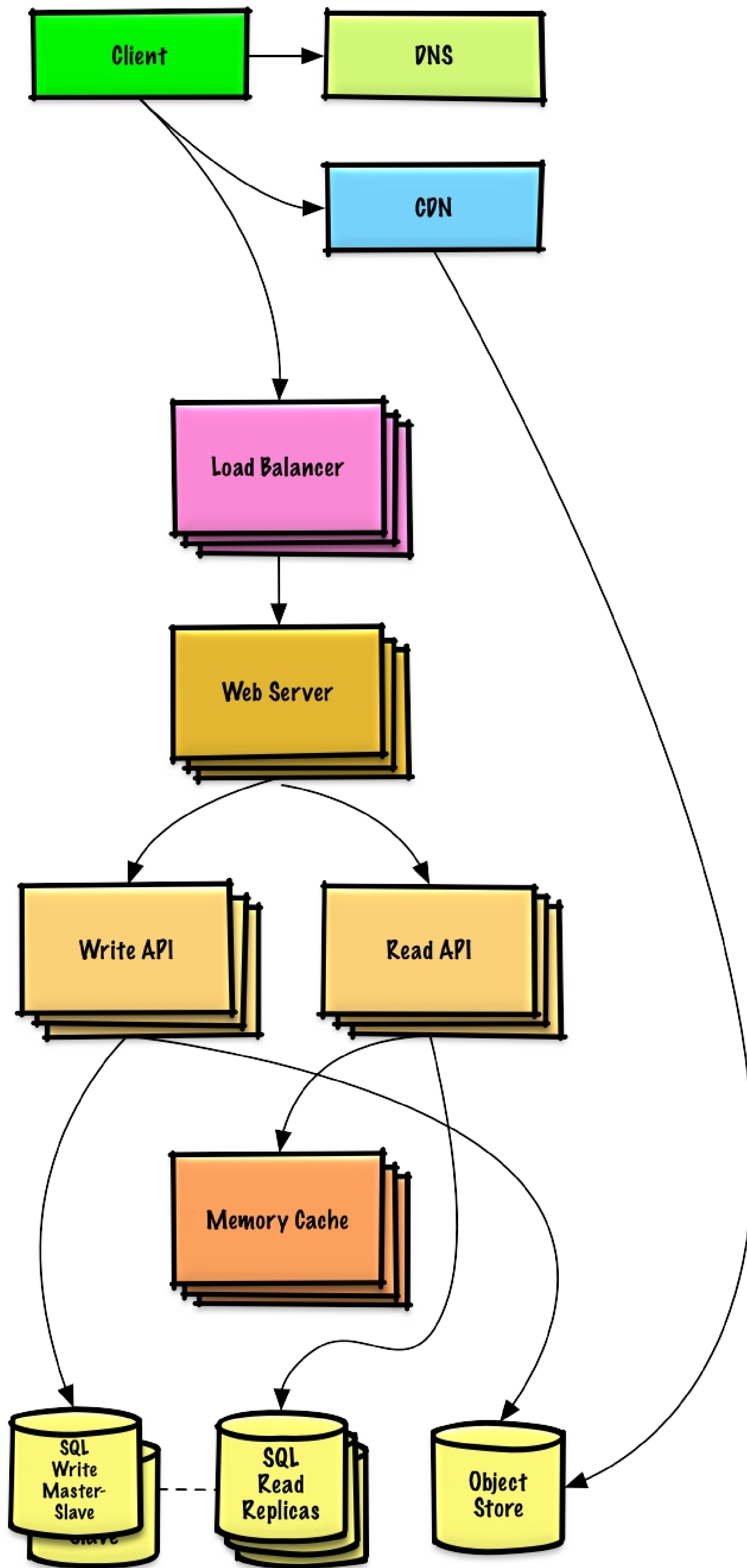
目标

- 下面的目标试图用 **Web**服务器 解决扩展问题
    - 基于 基准/负载测试 和 分析，你可能只需要实现其中的一两个技术
- 使用 横向扩展 来处理增加的负载和单点故障
    - 添加 负载均衡器 例如 Amazon 的 ELB 或 HAProxy
        - ELB 是高可用的
        - 如果你正在配置自己的 负载均衡器, 在多个可用区域中设置多台服务器用于 双活 或 主被 将提高可用性
        - 终止在 负载平衡器 上的SSL，以减少后端服务器上的计算负载，并简化证书管理
    - 在多个可用区域中使用多台 **Web**服务器
    - 在多个可用区域的 主-从 故障转移 模式中使用多个 **MySQL** 实例来改进冗余
- 分离 **Web** 服务器 和 应用服务器
    - 独立扩展和配置每一层
    - **Web** 服务器 可以作为 反向代理
    - 例如, 你可以添加 应用服务器 处理 读 **API** 而另外一些处理 写 **API**
- 将静态（和一些动态）内容转移到 内容分发网络 **(CDN)** 例如 CloudFront 以减少负载和延迟

折中方案, 可选方案, 和其他细节:

- 查阅以上链接获得更多细节

用户 +++


Client
DNS
CDN
Load Balancer
Web Server
Write API
Read API
Memory Cache
SQL
Write
Master-
Slave
SQL
Read
Replicas
Object
Store

注意： 内部负载均衡 不显示以减少混乱

假设

我们的 性能/负载测试 和 性能测试 显示我们读操作频繁（100:1 的读写比率），并且数据库在高读请求时表现很糟糕。

目标

- 下面的目标试图解决 **MySQL**数据库 的伸缩性问题
  -
    - 基于 基准/负载测试 和 分析，你可能只需要实现其中的一两个技术
- 将下列数据移动到一个 内存缓存，例如弹性缓存，以减少负载和延迟：
  - **MySQL** 中频繁访问的内容
    - 首先, 尝试配置 **MySQL** 数据库 缓存以查看是否足以在实现 内存缓存 之前缓解瓶颈
  - 来自 **Web** 服务器 的会话数据
    - **Web** 服务器 变成无状态的, 允许 自动伸缩
  - 从内存中读取 1 MB 内存需要大约 250 微秒，而从SSD中读取时间要长 4 倍，从磁盘读取的时间要长 80 倍。[1]
- 添加 **MySQL** 读取副本 来减少写主线程的负载
- 添加更多 **Web** 服务器 and 应用服务器 来提高响应

折中方案, 可选方案, 和其他细节:

- 查阅以上链接获得更多细节

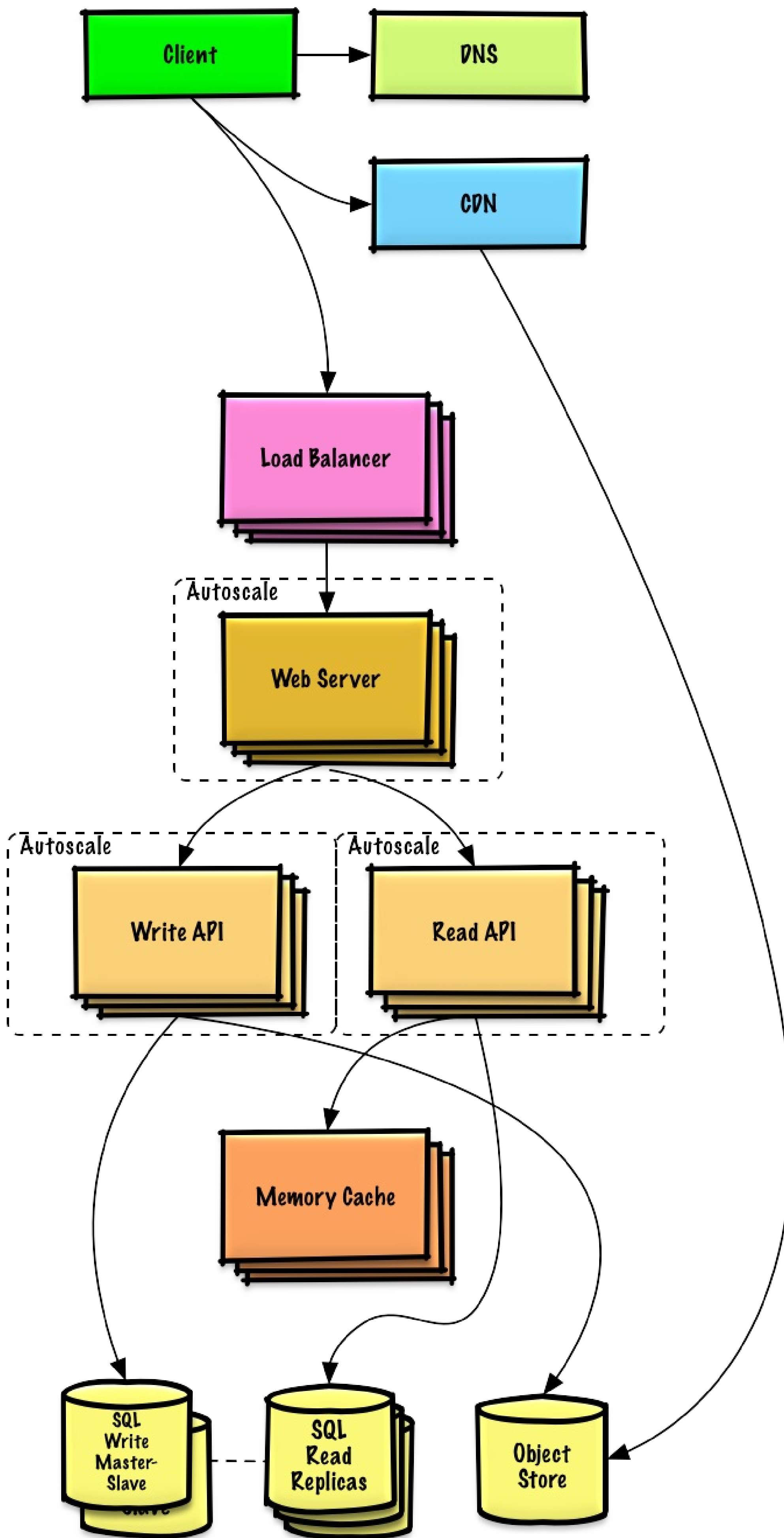
添加 **MySQL** 读取副本

- 除了添加和扩展 内存缓存，**MySQL** 读副本服务器 也能够帮助缓解在 **MySQL** 写主服务器 的负载。
- 添加逻辑到 **Web** 服务器 来区分读和写操作
- 在 **MySQL** 读副本服务器 之上添加 负载均衡器 （不是为了减少混乱）
- 大多数服务都是读取负载大于写入负载

折中方案, 可选方案, 和其他细节:

- 查阅 关系型数据库管理系统 (RDBMS) 章节

用户++++


Client
DNS
CDN
Load Balancer
Autoscale
Web Server
Autoscale
Write API
Autoscale
Read API
Memory Cache
SQL
Write
Master-
Slave
SQL
Read
Replicas
Object
Store

假设

基准/负载测试 和 分析 显示，在美国，正常工作时间存在流量峰值，当用户离开办公室时，流量骤降。我们认为，可以通过真实负载自动转换服务器数量来降低成本。我们是一家小商店，所以我们希望 DevOps 尽量自动化地进行 自动伸缩 和通用操作。

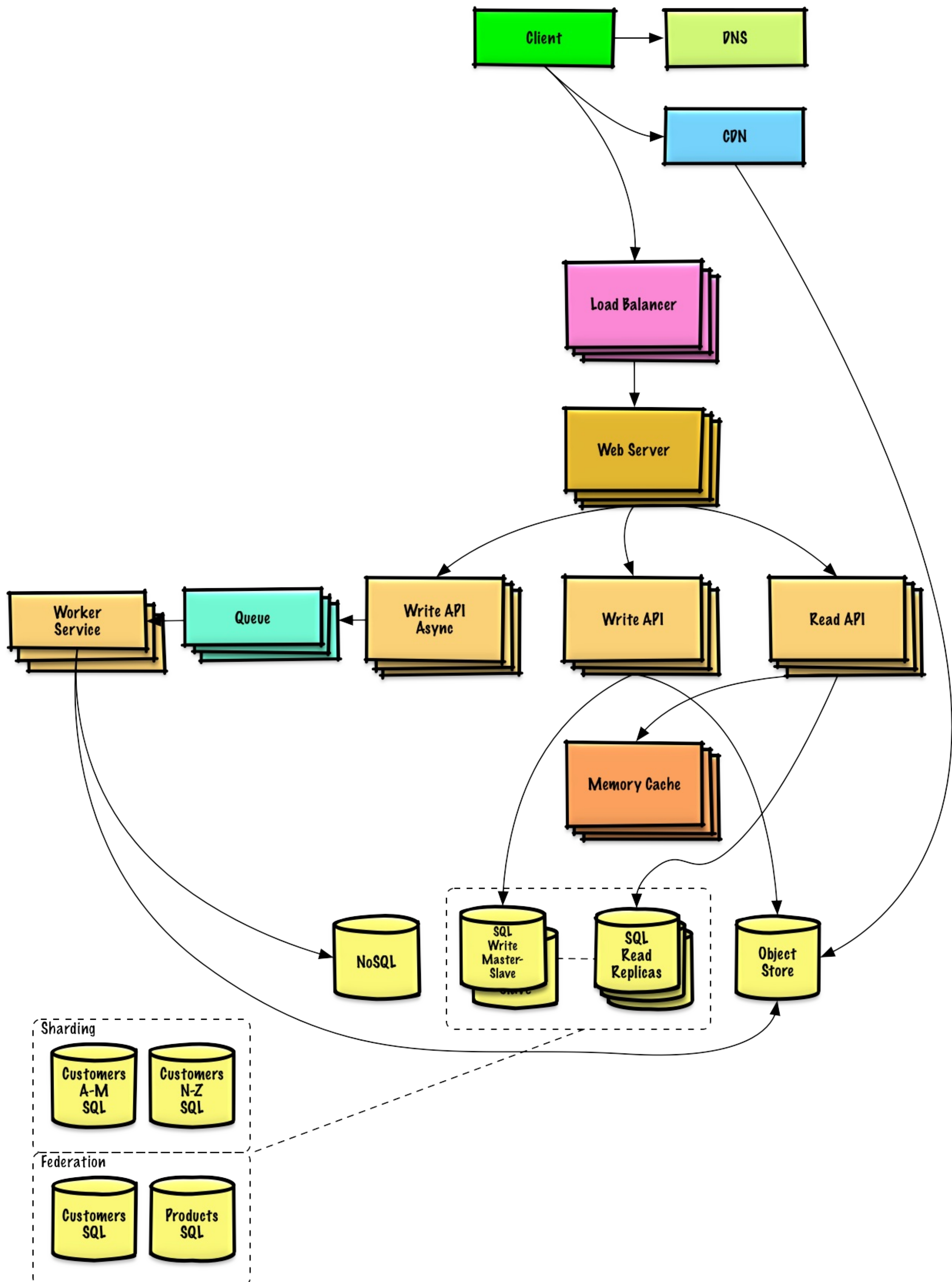
目标

- 根据需要添加 自动扩展
    - 跟踪流量高峰
    - 通过关闭未使用的实例来降低成本
- 自动化 DevOps
    - Chef, Puppet, Ansible 工具等
- 继续监控指标以解决瓶颈
    - 主机水平 - 检查一个 EC2 实例
    - 总水平 - 检查负载均衡器统计数据
    - 日志分析 - CloudWatch, CloudTrail, Loggly, Splunk, Sumo
    - 外部站点的性能 - Pingdom or New Relic
    - 处理通知和事件 - PagerDuty
    - 错误报告 - Sentry

添加自动扩展

- 考虑使用一个托管服务，比如AWS 自动扩展
    - 为每个 **Web** 服务器 创建一个组，并为每个 应用服务器 类型创建一个组，将每个组放置在多个可用区域中
    - 设置最小和最大实例数
    - 通过 CloudWatch 来扩展或收缩
        - 可预测负载的简单时间度量
        - 一段时间内的指标：
            - CPU 负载
            - 延迟
            - 网络流量
            - 自定义指标
    - 缺点
        - 自动扩展会引入复杂性
        - 可能需要一段时间才能适当扩大规模，以满足增加的需求，或者在需求下降时缩减规模

用户+++++



注释： 自动伸缩 组不显示以减少混乱

假设

当服务继续向着限制条件概述的方向发展，我们反复地运行 基准/负载测试 和 分析 来进一步发现和定位新的瓶颈。

目标

由于问题的约束，我们将继续提出扩展性的问题：

- 如果我们的 **MySQL** 数据库 开始变得过于庞大, 我们可能只考虑把数据在数据库中存储一段有限的时间, 同时在例如 Redshift 这样的数据仓库中存储其余的数据
  - 像 Redshift 这样的数据仓库能够轻松处理每月 1TB 的新内容
- 平均每秒 40,000 次的读取请求, 可以通过扩展 内存缓存 来处理热点内容的读取流量，这对于处理不均匀分布的流量和流量峰值也很有用
  - **SQL**读取副本 可能会遇到处理缓存未命中的问题, 我们可能需要使用额外的 SQL 扩展模式
- 对于单个 **SQL** 写主-从 模式来说，平均每秒 400 次写操作（明显更高）可能会很困难，同时还需要更多的扩展技术

SQL 扩展模型包括：

- 集合
- 分片
- 反范式
- SQL 调优

为了进一步处理高读和写请求，我们还应该考虑将适当的数据移动到一个 **NoSQL**数据库 ，例如 DynamoDB。

我们可以进一步分离我们的 应用服务器 以允许独立扩展。不需要实时完成的批处理任务和计算可以通过 Queues 和 Workers 异步完成：

- 以照片服务为例，照片上传和缩略图的创建可以分开进行
  - 客户端 上传图片
  - 应用服务器 推送一个任务到 队列 例如 SQS
  - EC2 上的 **Worker** 服务 或者 Lambda 从 队列 拉取 work，然后：
    - 创建缩略图
    - 更新 数据库
    - 在 对象存储 中存储缩略图

折中方案, 可选方案, 和其他细节:

- 查阅以上链接获得更多细节

额外的话题

> 根据问题的范围和剩余时间，还需要深入讨论其他问题。

SQL 扩展模式

- 读取副本
- 集合
- 分区
- 反规范化
- SQL 调优

NoSQL

- 键值存储
- 文档存储
- 宽表存储
- 图数据库
- SQL vs NoSQL

缓存

- 缓存到哪里
    - 客户端缓存
    - CDN 缓存
    - Web 服务缓存
    - 数据库缓存
    - 应用缓存
- 缓存什么
    - 数据库请求层缓存
    - 对象层缓存
- 何时更新缓存
    - 预留缓存
    - 完全写入
    - 延迟写 (写回)
    - 事先更新

异步性和微服务

- 消息队列

* 任务队列
* 回退压力
* 微服务

沟通

* 关于折中方案的讨论:
    * 客户端的外部通讯 - 遵循 REST 的 HTTP APIs
    * 内部通讯 - RPC
* 服务探索

安全性

参考 安全章节

延迟数字指标

查阅 每个程序员必懂的延迟数字

正在进行

* 继续基准测试并监控你的系统以解决出现的瓶颈问题
* 扩展是一个迭代的过程

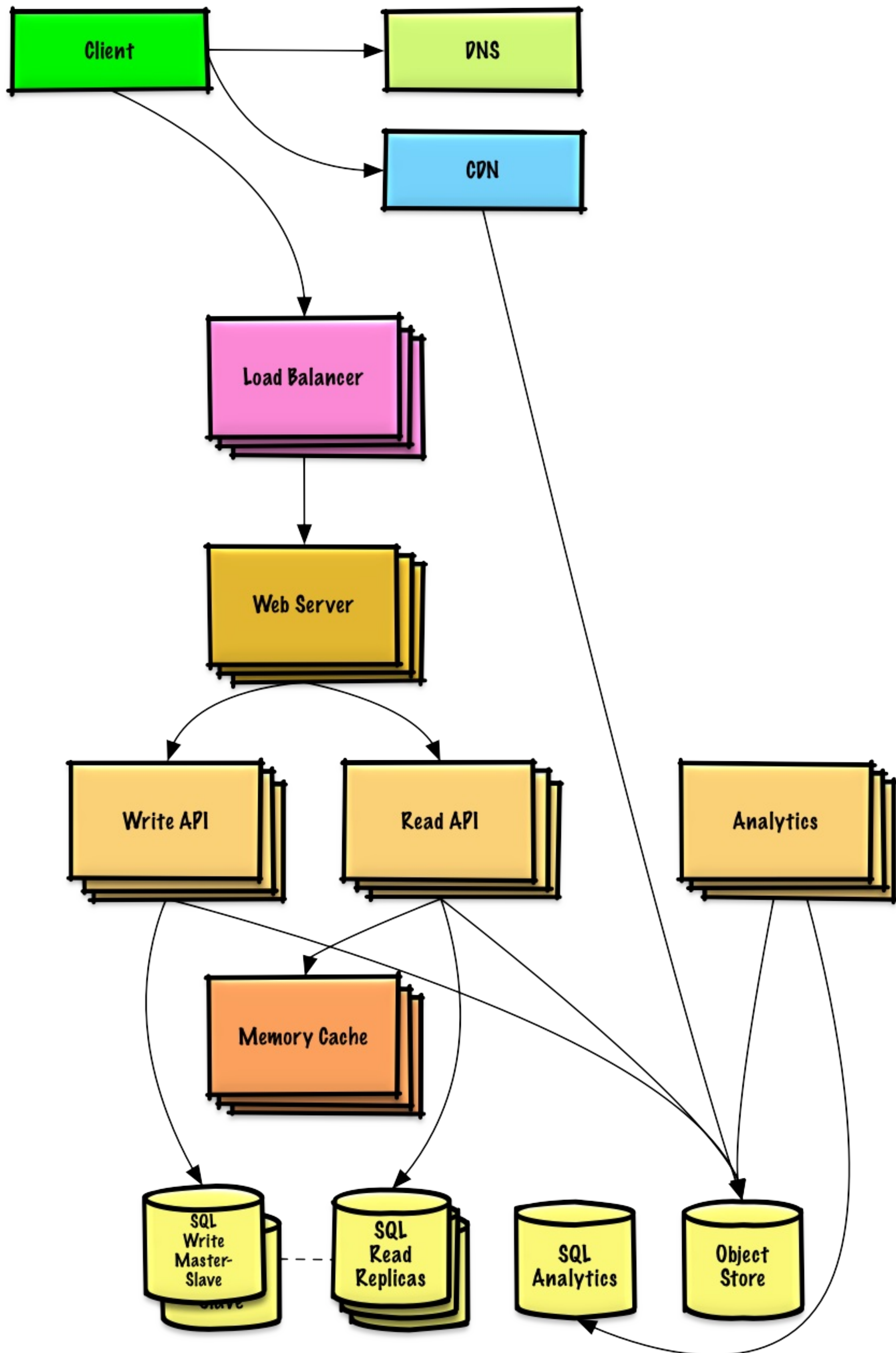
系统设计的面试题和解答

> 普通的系统设计面试题和相关事例的论述，代码和图表。
>
> 与内容有关的解答在 `solutions/` 文件夹中。

问题	
设计 Pastebin.com (或者 Bit.ly)	解答
设计 Twitter 时间线和搜索 (或者 Facebook feed 和搜索)	解答
设计一个网页爬虫	解答
设计 Mint.com	解答
为一个社交网络设计数据结构	解答
为搜索引擎设计一个 key-value 储存	解答
通过分类特性设计 Amazon 的销售排名	解答
在 AWS 上设计一个百万用户级别的系统	解答
添加一个系统设计问题	贡献

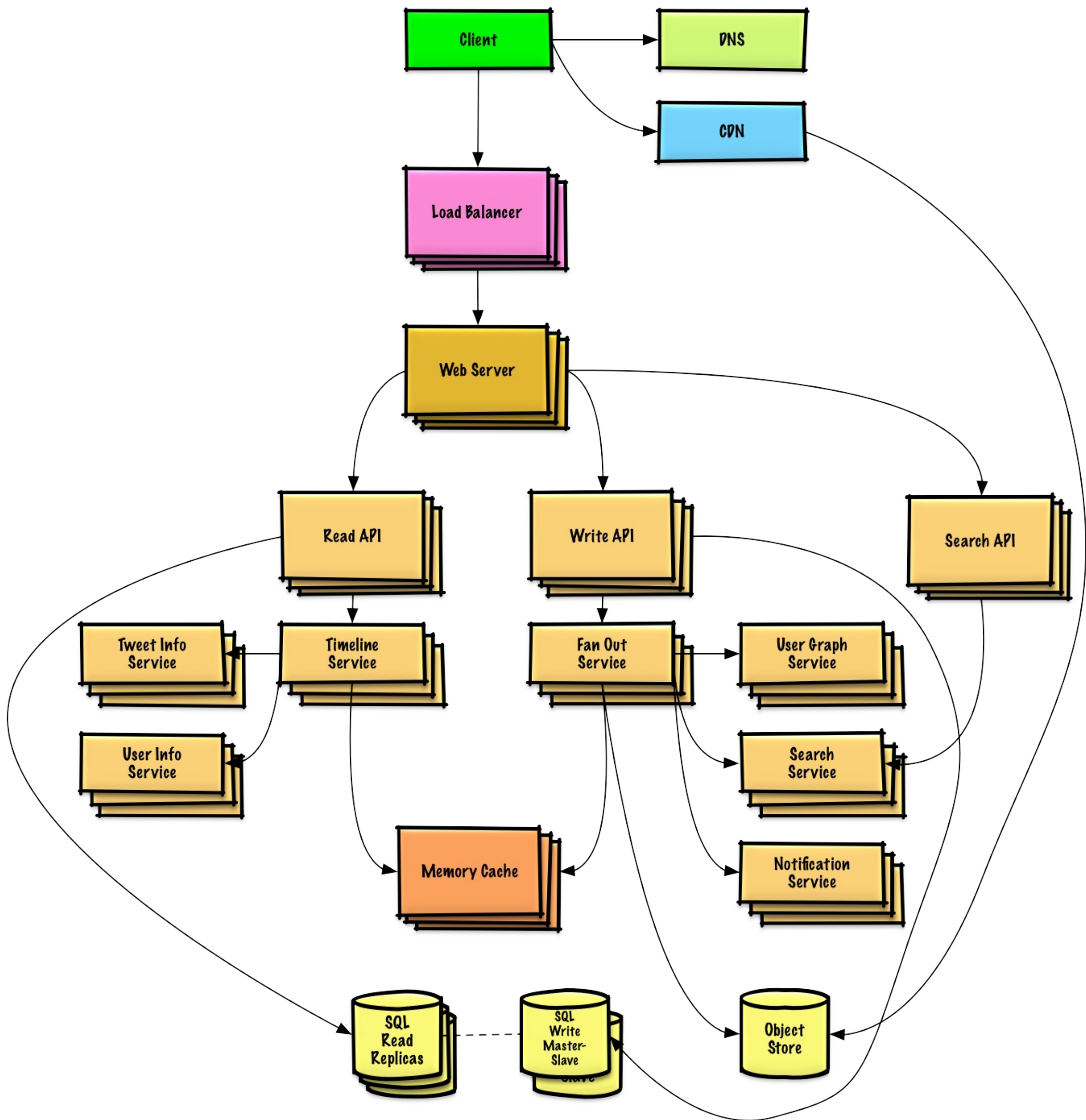
设计 Pastebin.com (或者 Bit.ly)

查看实践与解答



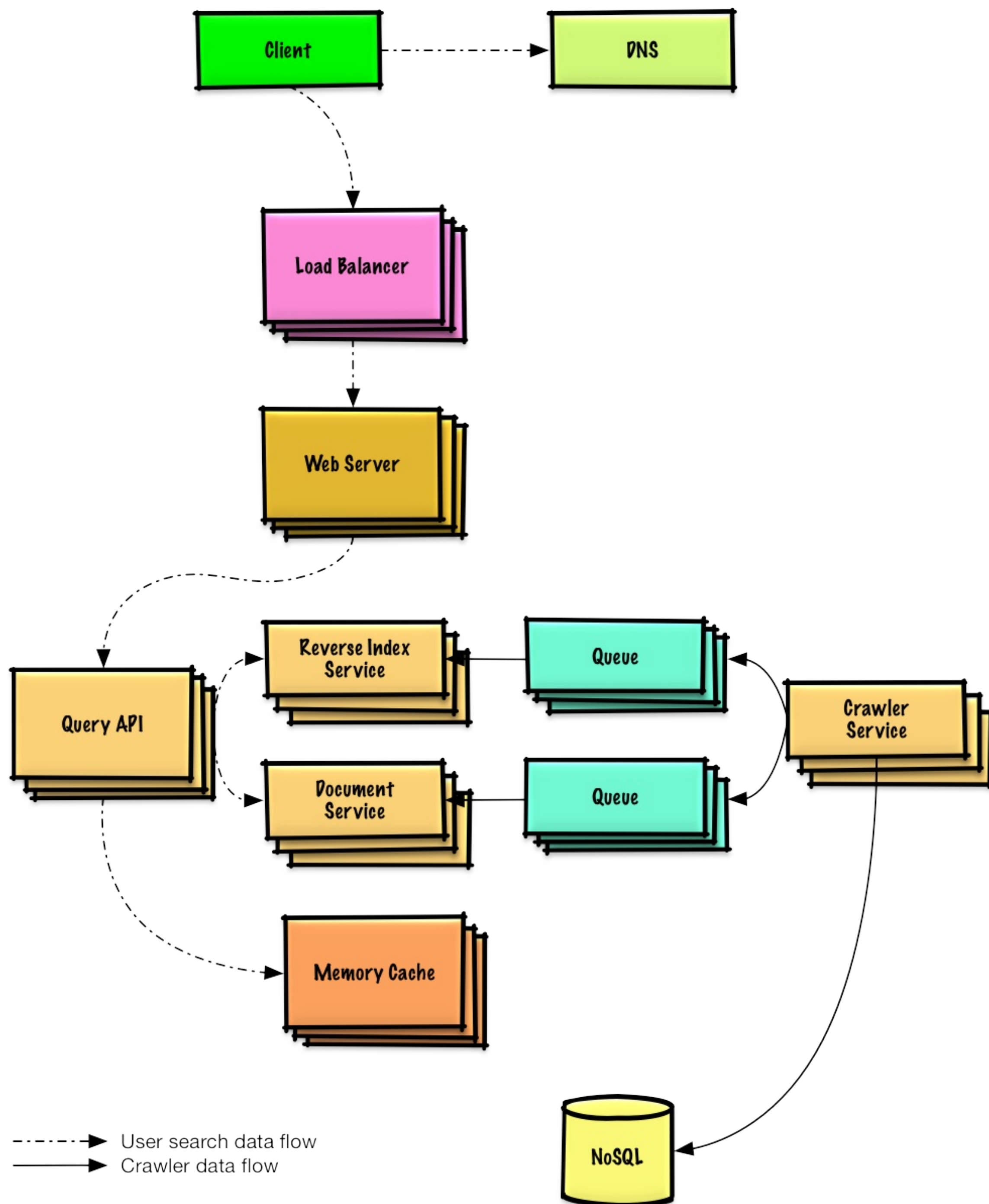
设计 Twitter 时间线和搜索 (或者 Facebook feed 和搜索)

查看实践与解答



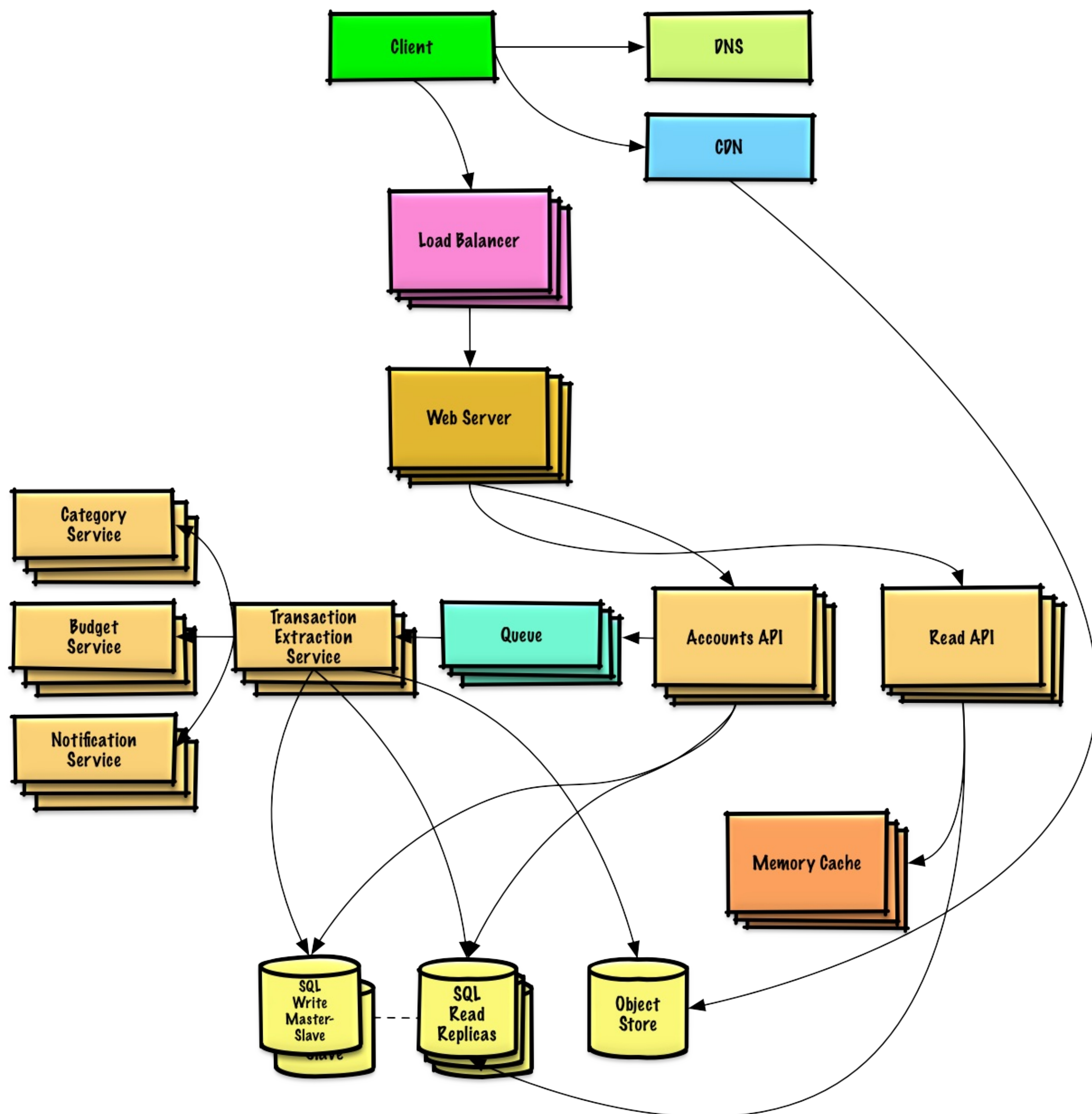
设计一个网页爬虫

查看实践与解答



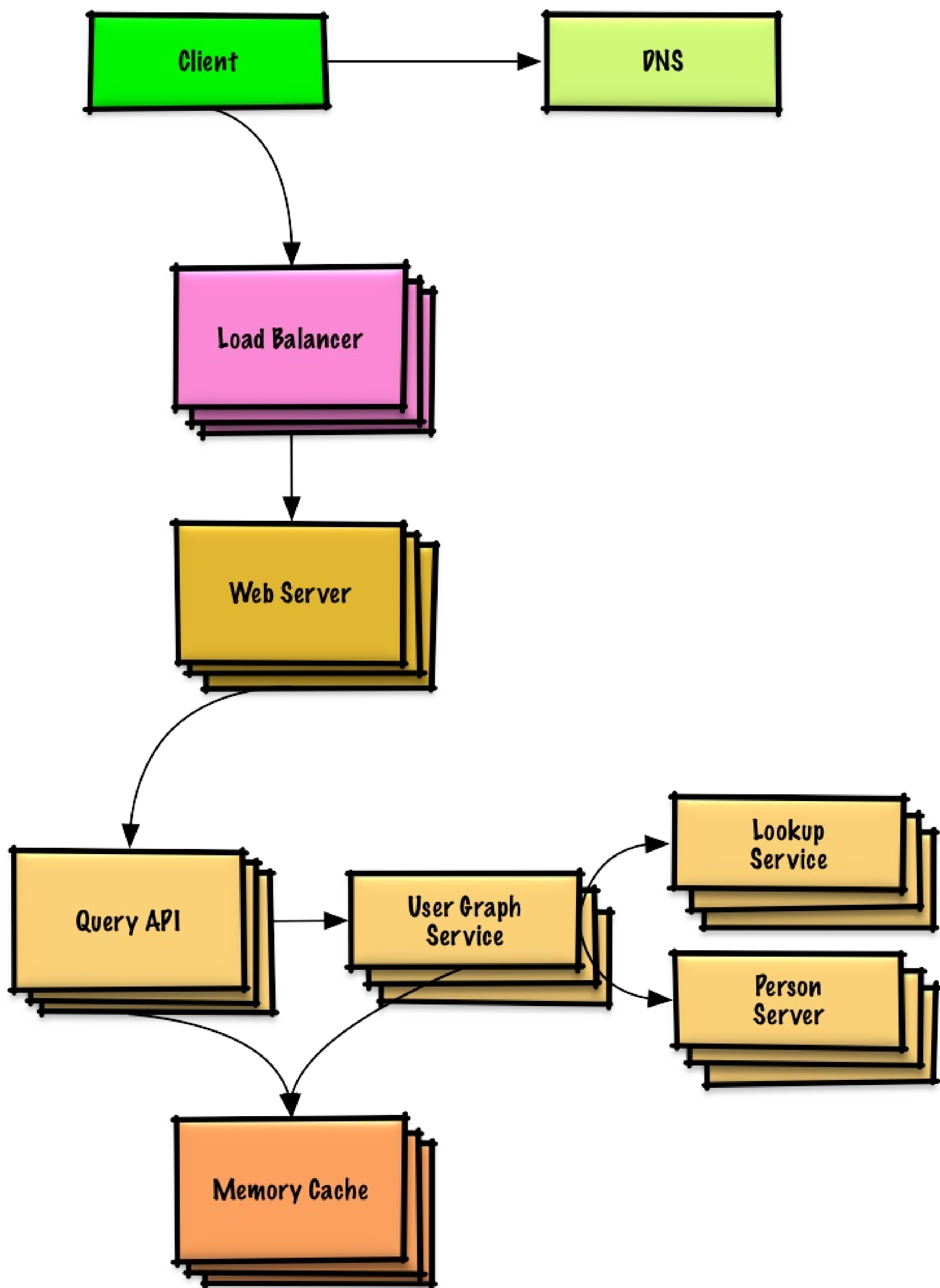
设计 Mint.com

查看实践与解答



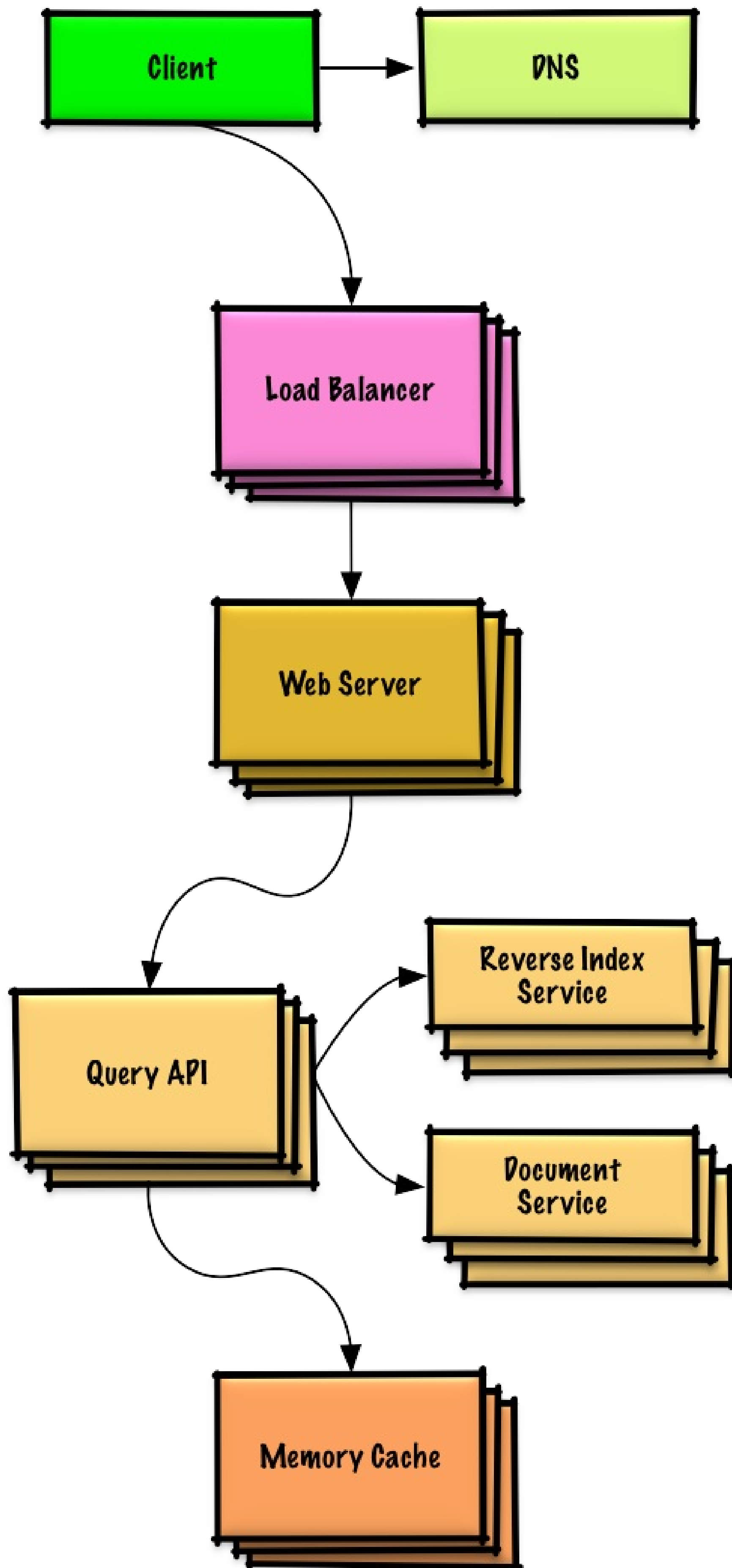
为一个社交网络设计数据结构

查看实践与解答



为搜索引擎设计一个 key-value 储存

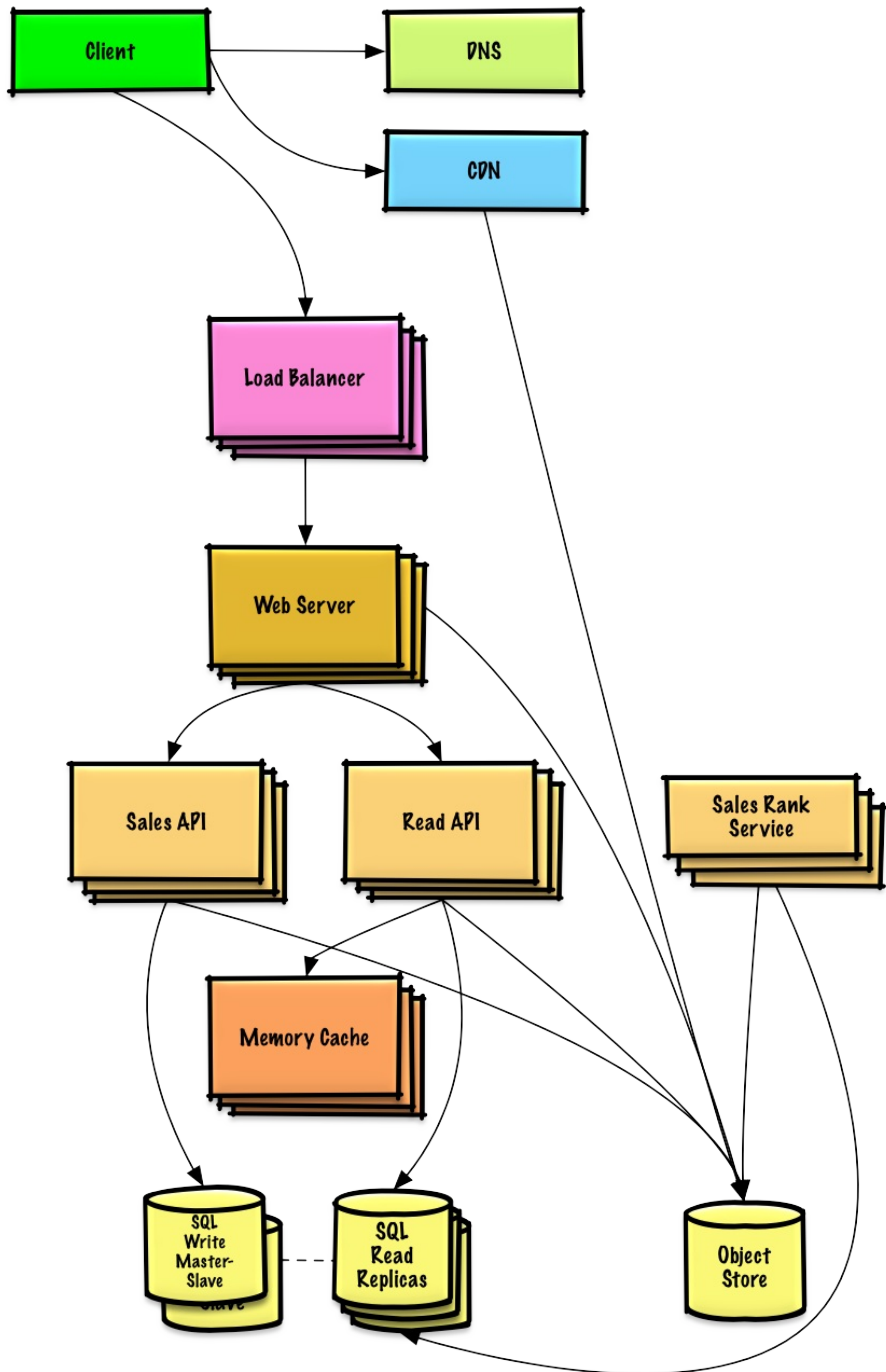
查看实践与解答



Client
DNS
Load Balancer
Web Server
Query API
Reverse Index Service
Document Service
Memory Cache

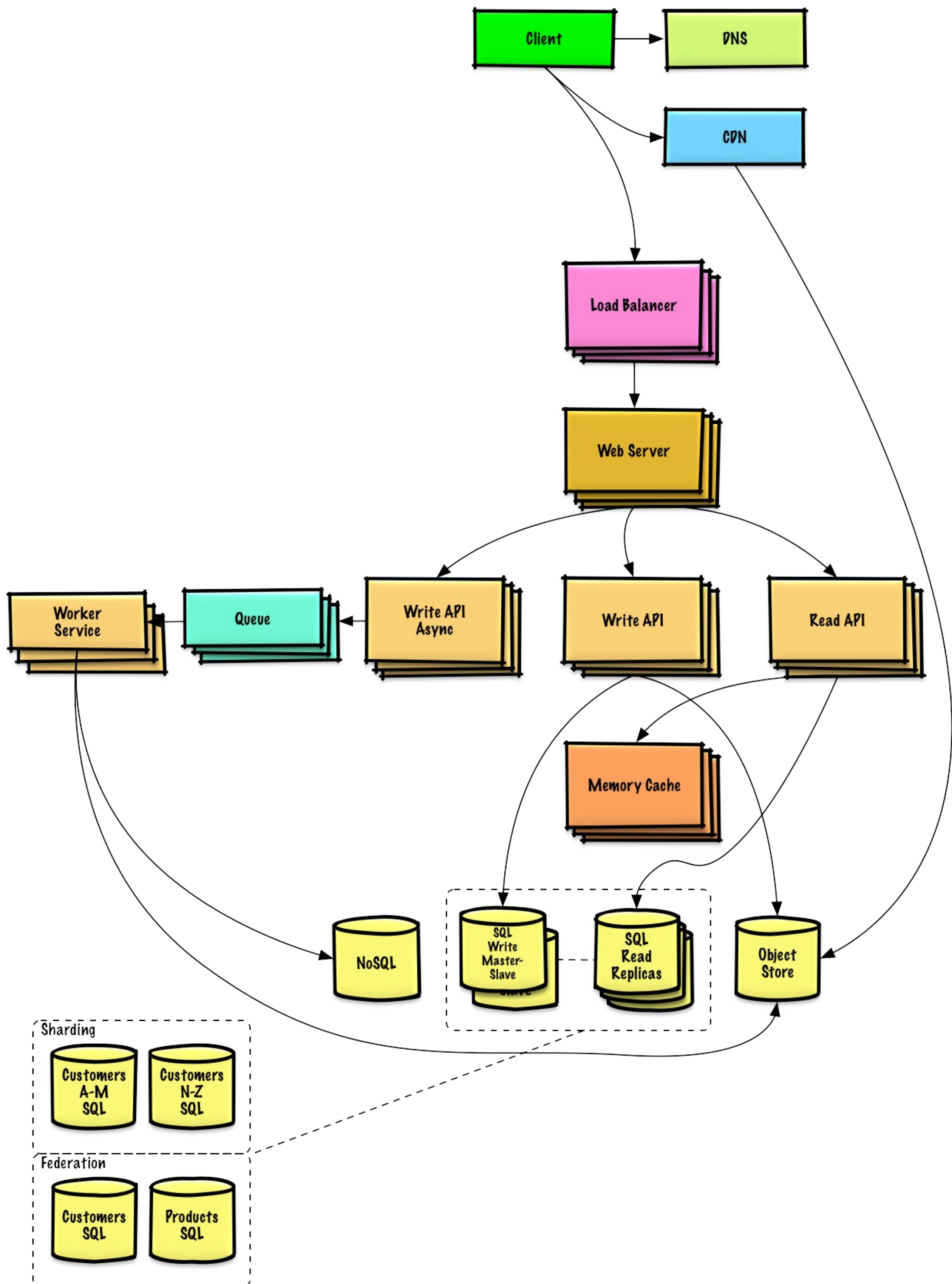
设计按类别分类的 Amazon 销售排名

查看实践与解答


Client
DNS
CDN
Load Balancer
Web Server
Sales API
Read API
Sales Rank Service
Memory Cache
SQL Write Master-Slave
SQL Read Replicas
Object Store

在 AWS 上设计一个百万用户级别的系统

查看实践与解答



面向对象设计的面试问题及解答

> 常见面向对象设计面试问题及实例讨论，代码和图表演示。
>
> 与内容相关的解决方案在 `solutions/` 文件夹中。
>
> 注：此节还在完善中

问题	
设计 hash map	解决方案
设计 LRU 缓存	解决方案
设计一个呼叫中心	解决方案
设计一副牌	解决方案
设计一个停车场	解决方案
设计一个聊天服务	解决方案
设计一个环形数组	待解决
添加一个面向对象设计问题	待解决

系统设计主题：从这里开始

不熟悉系统设计？

首先，你需要对一般性原则有一个基本的认识，知道它们是什么，怎样使用以及利弊。

第一步：回顾可扩展性（scalability）的视频讲座

哈佛大学可扩展性讲座

- 主题涵盖
  - 垂直扩展（Vertical scaling）
  - 水平扩展（Horizontal scaling）
  - 缓存
  - 负载均衡
  - 数据库复制
  - 数据库分区

第二步：回顾可扩展性文章

可扩展性

- 主题涵盖：
  - Clones
  - 数据库
  - 缓存
  - 异步

接下来的步骤

接下来，我们将看看高阶的权衡和取舍:

- 性能与可扩展性
- 延迟与吞吐量
- 可用性与一致性

记住每个方面都面临取舍和权衡。

然后，我们将深入更具体的主题，如 DNS、CDN 和负载均衡器。

性能与可扩展性

如果服务性能的增长与资源的增加是成比例的，服务就是可扩展的。通常，提高性能意味着服务于更多的工作单元，另一方面，当数据集增长时，同样也可以处理更大的工作单位。[1]

另一个角度来看待性能与可扩展性:

- 如果你的系统有性能问题，对于单个用户来说是缓慢的。
- 如果你的系统有可扩展性问题，单个用户较快但在高负载下会变慢。

来源及延伸阅读

- 简单谈谈可扩展性
- 可扩展性，可用性，稳定性和模式

延迟与吞吐量

延迟是执行操作或运算结果所花费的时间。

吞吐量是单位时间内（执行）此类操作或运算的数量。

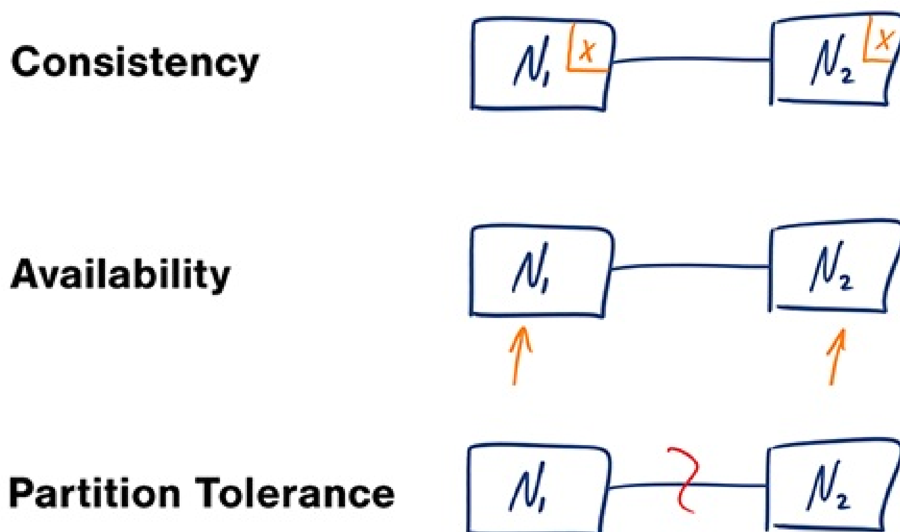
通常，你应该以可接受级延迟下最大化吞吐量为目标。

来源及延伸阅读

- 理解延迟与吞吐量

可用性与一致性

CAP 理论



来源：再看 CAP 理论

在一个分布式计算系统中，只能同时满足下列的两点:

- 一致性 ─ 每次访问都能获得最新数据但可能会收到错误响应
- 可用性 ─ 每次访问都能收到非错响应，但不保证获取到最新数据
- 分区容错性 ─ 在任意分区网络故障的情况下系统仍能继续运行

网络并不可靠，所以你应要支持分区容错性，并需要在软件可用性和一致性间做出取舍。

CP ─ 一致性和分区容错性

等待分区节点的响应可能会导致延时错误。如果你的业务需求需要原子读写，CP 是一个不错的选择。

AP ─ 可用性与分区容错性

响应节点上可用数据的最近版本可能并不是最新的。当分区解析完后，写入（操作）可能需要一些时间来传播。

如果业务需求允许最终一致性，或当有外部故障时要求系统继续运行，AP 是一个不错的选择。

来源及延伸阅读

- 再看 CAP 理论
- 通俗易懂地介绍 CAP 理论
- CAP FAQ

一致性模式

有同一份数据的多份副本，我们面临着怎样同步它们的选择，以便让客户端有一致的显示数据。回想 CAP 理论中的一致性定义 ─ 每次访问都能获得最新数据但可能会收到错误响应

弱一致性

在写入之后，访问可能看到，也可能看不到（写入数据）。尽力优化之让其能访问最新数据。

这种方式可以 memcached 等系统中看到。弱一致性在 VoIP，视频聊天和实时多人游戏等真实用例中表现不错。打个比方，如果你在通话中丢失信号几秒钟时间，当重新连接时你是听不到这几秒钟所说的话的。

最终一致性

在写入后，访问最终能看到写入数据（通常在数毫秒内）。数据被异步复制。

DNS 和 email 等系统使用的是此种方式。最终一致性在高可用性系统中效果不错。

强一致性

在写入后，访问立即可见。数据被同步复制。

文件系统和关系型数据库（RDBMS）中使用的是此种方式。强一致性在需要记录的系统中运作良好。

来源及延伸阅读

- Transactions across data centers

可用性模式

有两种支持高可用性的模式: 故障切换（**fail-over**）和复制（**replication**）。

故障切换

工作到备用切换（Active-passive）

关于工作到备用的故障切换流程是，工作服务器发送周期信号给待机中的备用服务器。如果周期信号中断，备用服务器切换成工作服务器的 IP 地址并恢复服务。

宕机时间取决于备用服务器处于“热”待机状态还是需要从“冷”待机状态进行启动。只有工作服务器处理流量。

工作到备用的故障切换也被称为主从切换。

双工作切换（Active-active）

在双工作切换中，双方都在管控流量，在它们之间分散负载。

如果是外网服务器，DNS 将需要对两方都了解。如果是内网服务器，应用程序逻辑将需要对两方都了解。

双工作切换也可以称为主主切换。

缺陷：故障切换

- 故障切换需要添加额外硬件并增加复杂性。
- 如果新写入数据在能被复制到备用系统之前，工作系统出现了故障，则有可能会丢失数据。

复制

主─从复制和主─主复制

这个主题进一步探讨了数据库部分:

- 主─从复制
- 主─主复制

域名系统

来源：**DNS** 安全介绍

域名系统是把 www.example.com 等域名转换成 IP 地址。

域名系统是分层次的，一些 DNS 服务器位于顶层。当查询（域名） IP 时，路由或 ISP 提供连接 DNS 服务器的信息。较底层的 DNS 服务器缓存映射，它可能会因为 DNS 传播延时而失效。DNS 结果可以缓存在浏览器或操作系统中一段时间，时间长短取决于存活时间 TTL。

- **NS** 记录（域名服务） — 指定解析域名或子域名的 DNS 服务器。
- **MX** 记录（邮件交换） — 指定接收信息的邮件服务器。
- **A** 记录（地址） — 指定域名对应的 IP 地址记录。
- **CNAME**（规范） — 一个域名映射到另一个域名或 `CNAME` 记录（ example.com 指向 www.example.com ）或映射到一个 `A` 记录。

CloudFlare 和 Route 53 等平台提供管理 DNS 的功能。某些 DNS 服务通过集中方式来路由流量:

- 加权轮询调度
  - 防止流量进入维护中的服务器
  - 在不同大小集群间负载均衡
  - A/B 测试

* 基于延迟路由
* 基于地理位置路由

缺陷:DNS

* 虽说缓存可以减轻 DNS 延迟，但连接 DNS 服务器还是带来了轻微的延迟。
* 虽然它们通常由政府，网络服务提供商和大公司管理，但 DNS 服务管理仍可能是复杂的。
* DNS 服务最近遭受 DDoS 攻击，阻止不知道 Twtter IP 地址的用户访问 Twiiter。

来源及延伸阅读

* DNS 架构.aspx)
* Wikipedia
* 关于 DNS 的文章

内容分发网络（CDN）



来源：为什么使用 CDN

内容分发网络（CDN）是一个全球性的代理服务器分布式网络，它从靠近用户的位置提供内容。通常，HTML/CSS/JS，图片和视频等静态内容由 CDN 提供，虽然亚马逊 CloudFront 等也支持动态内容。CDN 的 DNS 解析会告知客户端连接哪台服务器。

将内容存储在 CDN 上可以从两个方面来提供性能:

- 从靠近用户的数据中心提供资源
- 通过 CDN 你的服务器不必真的处理请求

CDN 推送（push）

当你服务器上内容发生变动时，推送 CDN 接受新内容。直接推送给 CDN 并重写 URL 地址以指向你的内容的 CDN 地址。你可以配置内容到期时间及何时更新。内容只有在更改或新增是才推送，流量最小化，但储存最大化。

CDN 拉取（pull）

CDN 拉取是当第一个用户请求该资源时，从服务器上拉取资源。你将内容留在自己的服务器上并重写 URL 指向 CDN 地址。直到内容被缓存在 CDN 上为止，这样请求只会更慢，

存活时间（TTL）决定缓存多久时间。CDN 拉取方式最小化 CDN 上的储存空间，但如果过期文件并在实际更改之前被拉取，则会导致冗余的流量。

高流量站点使用 CDN 拉取效果不错，因为只有最近请求的内容保存在 CDN 中，流量才能更平衡地分散。

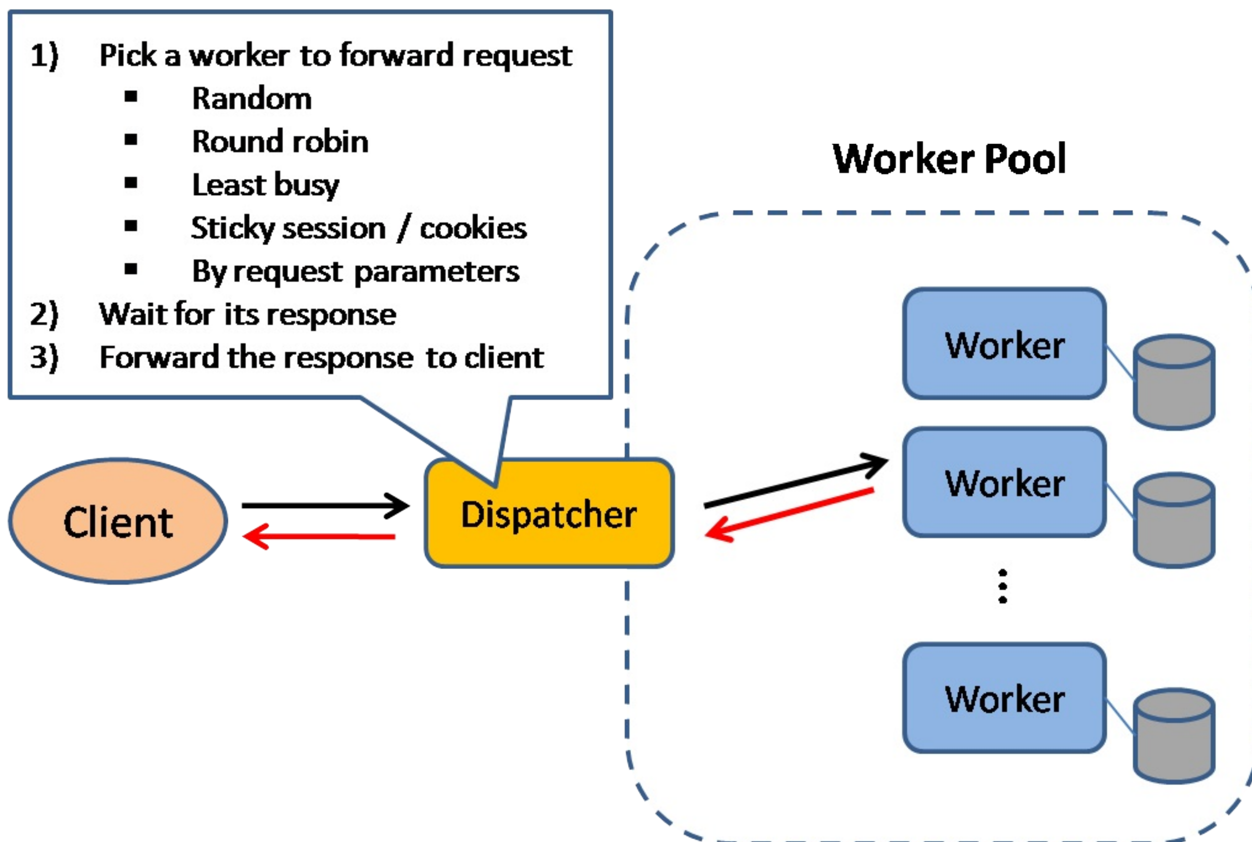
缺陷：CDN

- CDN 成本可能因流量而异，可能在权衡之后你将不会使用 CDN。
- 如果在 TTL 过期之前更新内容，CDN 缓存内容可能会过时。
- CDN 需要更改静态内容的 URL 地址以指向 CDN。

来源及延伸阅读

- 全球性内容分发网络
- CDN 拉取和 CDN 推送的区别
- Wikipedia

负载均衡器



来源：可扩展的系统设计模式

负载均衡器将传入的请求分发到应用服务器和数据库等计算资源。无论哪种情况，负载均衡器将从计算资源来的响应返回给恰当的客户端。负载均衡器的效用在于:

- 防止请求进入不好的服务器
- 防止资源过载
- 帮助消除单一的故障点

负载均衡器可以通过硬件（昂贵）或 HAProxy 等软件来实现。 增加的好处包括:

- **SSL** 终结 — 解密传入的请求并加密服务器响应，这样的话后端服务器就不必再执行这些潜在高消耗运算了。
    - 不需要再每台服务器上安装 X.509 证书。
- **Session** 留存 — 如果 Web 应用程序不追踪会话，发出 cookie 并将特定客户端的请求路由到同一实例。

通常会设置采用工作—备用 或 双工作 模式的多个负载均衡器，以免发生故障。

负载均衡器能基于多种方式来路由流量:

- 随机

- 最少负载
- Session/cookie
- 轮询调度或加权轮询调度算法
- 四层负载均衡
- 七层负载均衡

四层负载均衡

四层负载均衡根据监看传输层的信息来决定如何分发请求。通常，这会涉及来源，目标 IP 地址和请求头中的端口，但不包括数据包（报文）内容。四层负载均衡执行网络地址转换（NAT）来向上游服务器转发网络数据包。

七层负载均衡器

七层负载均衡器根据监控应用层来决定怎样分发请求。这会涉及请求头的内容，消息和 cookie。七层负载均衡器终结网络流量，读取消息，做出负载均衡判定，然后传送给特定服务器。比如，一个七层负载均衡器能直接将视频流量连接到托管视频的服务器，同时将更敏感的用户账单流量引导到安全性更强的服务器。

以损失灵活性为代价，四层负载均衡比七层负载均衡花费更少时间和计算资源，虽然这对现代商用硬件的性能影响甚微。

水平扩展

负载均衡器还能帮助水平扩展，提高性能和可用性。使用商业硬件的性价比更高，并且比在单台硬件上垂直扩展更贵的硬件具有更高的可用性。相比招聘特定企业系统人才，招聘商业硬件方面的人才更加容易。

缺陷：水平扩展

- 水平扩展引入了复杂度并涉及服务器复制
    - 服务器应该是无状态的:它们也不该包含像 session 或资料图片等与用户关联的数据。
    - session 可以集中存储在数据库或持久化缓存（Redis、Memcached）的数据存储区中。
- 缓存和数据库等下游服务器需要随着上游服务器进行扩展，以处理更多的并发连接。

缺陷：负载均衡器

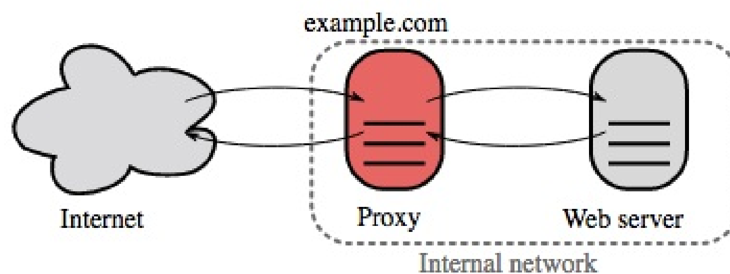
- 如果没有足够的资源配置或配置错误，负载均衡器会变成一个性能瓶颈。
- 引入负载均衡器以帮助消除单点故障但导致了额外的复杂性。

- 单个负载均衡器会导致单点故障，但配置多个负载均衡器会进一步增加复杂性。

来源及延伸阅读

- NGINX 架构
- HAProxy 架构指南
- 可扩展性
- Wikipedia)
- 四层负载平衡
- 七层负载平衡
- ELB 监听器配置

反向代理（web 服务器）



资料来源：维基百科

反向代理是一种可以集中地调用内部服务，并提供统一接口给公共客户的 web 服务器。来自客户端的请求先被反向代理服务器转发到可响应请求的服务器，然后代理再把服务器的响应结果返回给客户端。

带来的好处包括：

- 增加安全性 - 隐藏后端服务器的信息，屏蔽黑名单中的 IP，限制每个客户端的连接数。
- 提高可扩展性和灵活性 - 客户端只能看到反向代理服务器的 IP，这使你可以增减服务器或者修改它们的配置。
- 本地终结 **SSL** 会话 - 解密传入请求，加密服务器响应，这样后端服务器就不必完成这些潜在的高成本的操作。
    - 免除了在每个服务器上安装 X.509 证书的需要
- 压缩 - 压缩服务器响应
- 缓存 - 直接返回命中的缓存结果
- 静态内容 - 直接提供静态内容
    - HTML/CSS/JS
    - 图片
    - 视频
    - 等等

负载均衡器与反向代理

- 当你有多个服务器时，部署负载均衡器非常有用。通常，负载均衡器将流量路由给一组功能相同的服务器上。
- 即使只有一台 web 服务器或者应用服务器时，反向代理也有用，可以参考上一节介绍的好处。
- NGINX 和 HAProxy 等解决方案可以同时支持第七层反向代理和负载均衡。

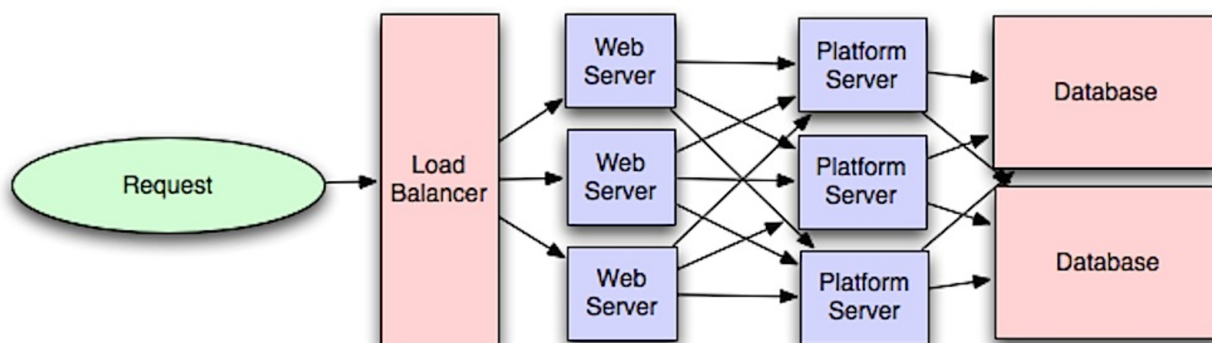
不利之处：反向代理

- 引入反向代理会增加系统的复杂度。
- 单独一个反向代理服务器仍可能发生单点故障，配置多台反向代理服务器（如故障转移）会进一步增加复杂度。

来源及延伸阅读

- 反向代理与负载均衡
- NGINX 架构
- HAProxy 架构指南
- Wikipedia

应用层



资料来源：可缩放系统构架介绍

将 Web 服务层与应用层（也被称作平台层）分离，可以独立缩放和配置这两层。添加新的 API 只需要添加应用服务器，而不必添加额外的 web 服务器。

单一职责原则提倡小型的，自治的服务共同合作。小团队通过提供小型的服务，可以更激进地计划增长。

应用层中的工作进程也有可以实现异步化。

微服务

与此讨论相关的话题是 微服务，可以被描述为一系列可以独立部署的小型的，模块化服务。每个服务运行在一个独立的线程中，通过明确定义的轻量级机制通讯，共同实现业务目标。[1]

例如，Pinterest 可能有这些微服务： 用户资料、关注者、Feed 流、搜索、照片上传等。

服务发现

像 Consul，Etcd 和 Zookeeper 这样的系统可以通过追踪注册名、地址、端口等信息来帮助服务互相发现对方。Health checks 可以帮助确认服务的完整性和是否经常使用一个 HTTP 路径。Consul 和 Etcd 都有一个内建的 key-value 存储 用来存储配置信息和其他的共享信息。

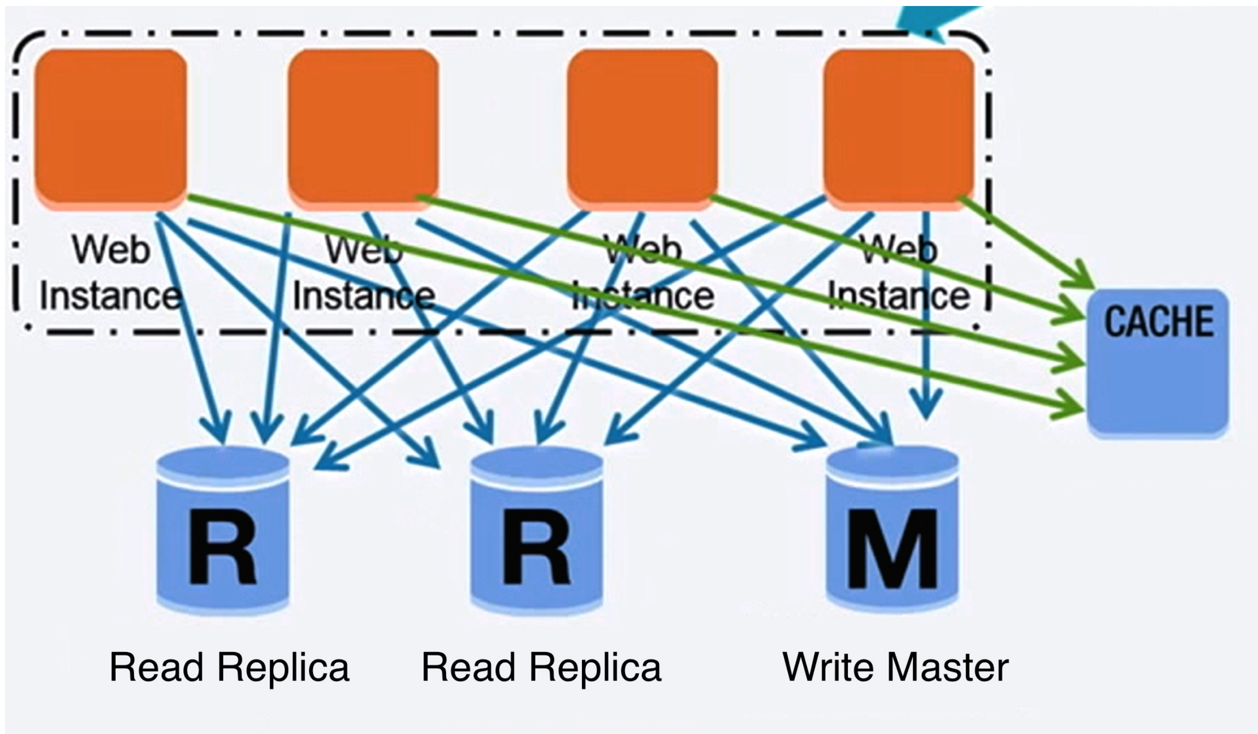
不利之处：应用层

- 添加由多个松耦合服务组成的应用层，从架构、运营、流程等层面来讲将非常不同（相对于单体系统）。
- 微服务会增加部署和运营的复杂度。

来源及延伸阅读

- 可缩放系统构架介绍
- 破解系统设计面试
- 面向服务架构
- Zookeeper 介绍
- 构建微服务，你所需要知道的一切

数据库



资料来源：扩展你的用户数到第一个一千万

关系型数据库管理系统（RDBMS）

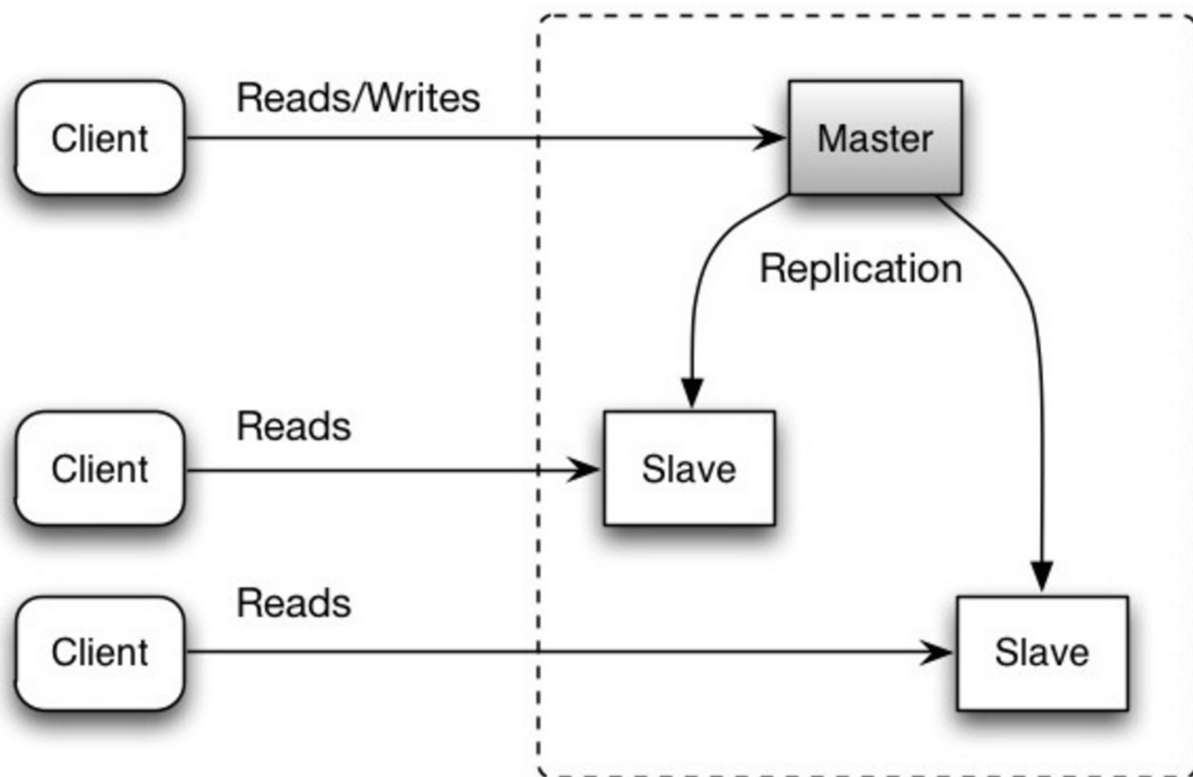
像 SQL 这样的关系型数据库是一系列以表的形式组织的数据项集合。

> 校对注：这里作者 SQL 可能指的是 MySQL

ACID 用来描述关系型数据库事务的特性。

- 原子性 - 每个事务内部所有操作要么全部完成，要么全部不完成。
- 一致性 - 任何事务都使数据库从一个有效的状态转换到另一个有效状态。
- 隔离性 - 并发执行事务的结果与顺序执行事务的结果相同。
- 持久性 - 事务提交后，对系统的影响是永久的。

关系型数据库扩展包括许多技术：主从复制、主主复制、联合、分片、非规范化和 **SQL**调优。



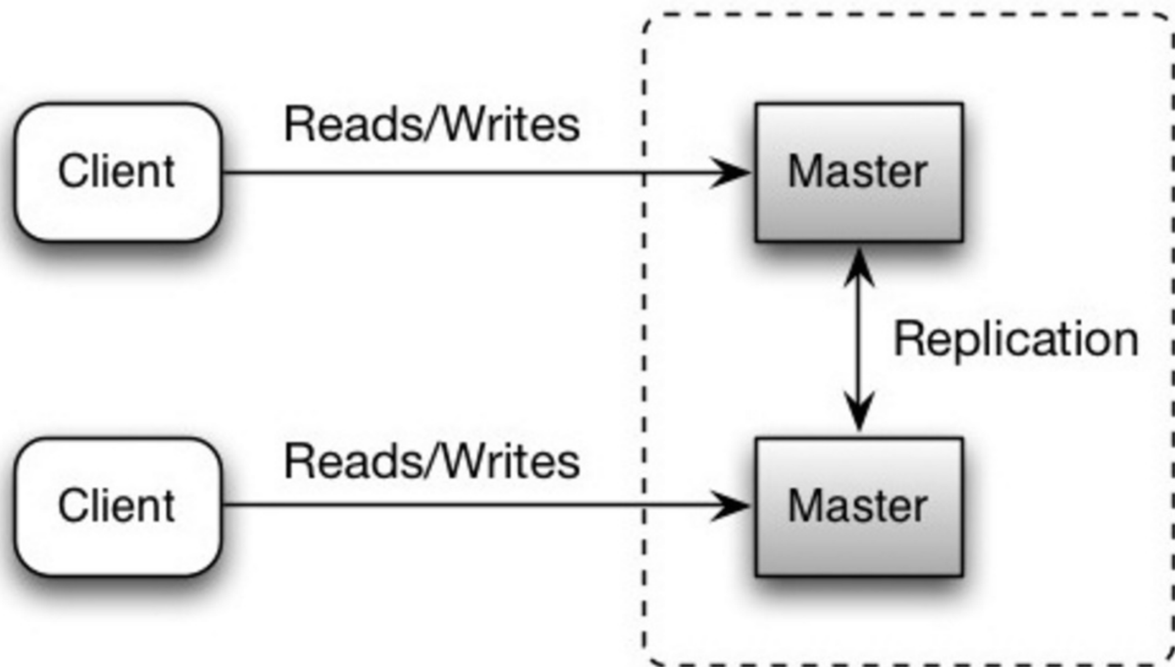
资料来源：可扩展性、可用性、稳定性、模式

主从复制

主库同时负责读取和写入操作，并复制写入到一个或多个从库中，从库只负责读操作。树状形式的从库再将写入复制到更多的从库中去。如果主库离线，系统可以以只读模式运行，直到某个从库被提升为主库或有新的主库出现。

不利之处：主从复制

- 将从库提升为主库需要额外的逻辑。
- 参考不利之处：复制中，主从复制和主主复制共同的问题。



资料来源：可扩展性、可用性、稳定性、模式

主主复制

两个主库都负责读操作和写操作，写入操作时互相协调。如果其中一个主库挂机，系统可以继续读取和写入。

不利之处： 主主复制

- 你需要添加负载均衡器或者在应用逻辑中做改动，来确定写入哪一个数据库。
- 多数主-主系统要么不能保证一致性（违反 ACID），要么因为同步产生了写入延迟。
- 随着更多写入节点的加入和延迟的提高，如何解决冲突显得越发重要。
- 参考不利之处：复制中，主从复制和主主复制共同的问题。

不利之处：复制

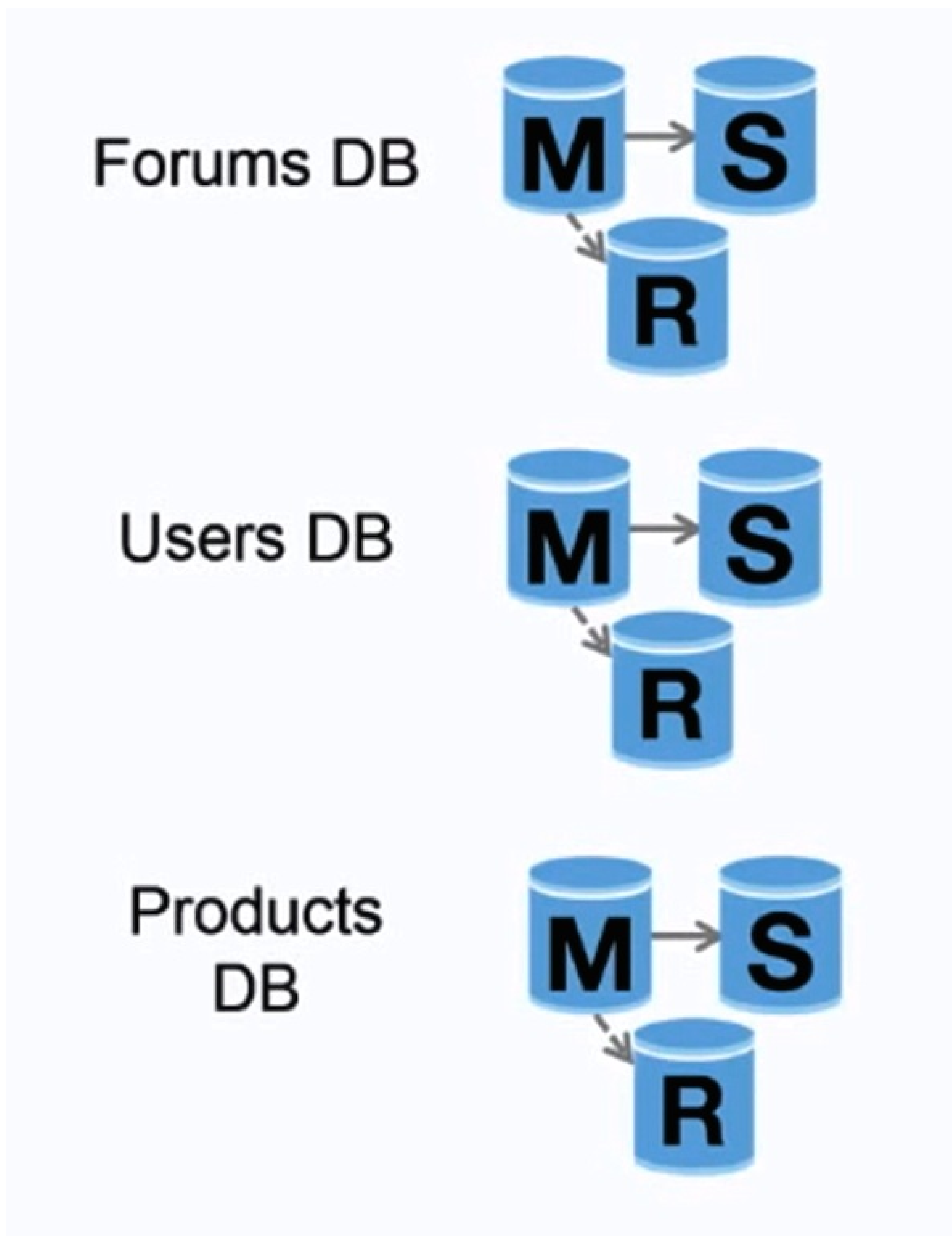
- 如果主库在将新写入的数据复制到其他节点前挂掉，则有数据丢失的可能。
- 写入会被重放到负责读取操作的副本。副本可能因为过多写操作阻塞住，导致读取功能异常。
- 读取从库越多，需要复制的写入数据就越多，导致更严重的复制延迟。
- 在某些数据库系统中，写入主库的操作可以用多个线程并行写入，但读取副本只支持单线程顺序地写入。
- 复制意味着更多的硬件和额外的复杂度。

来源及延伸阅读

- 扩展性，可用性，稳定性模式

- 多主复制

联合



资料来源：扩展你的用户数到第一个一千万

联合（或按功能划分）将数据库按对应功能分割。例如，你可以有三个数据库：论坛、用户和产品，而不仅是一个单体数据库，从而减少每个数据库的读取和写入流量，减少复制延迟。较小的数据库意味着更多适合放入内存的数据，进而意味着更高的缓存命中几率。没有只能串行写入的中心化主库，你可以并行写入，提高负载能力。

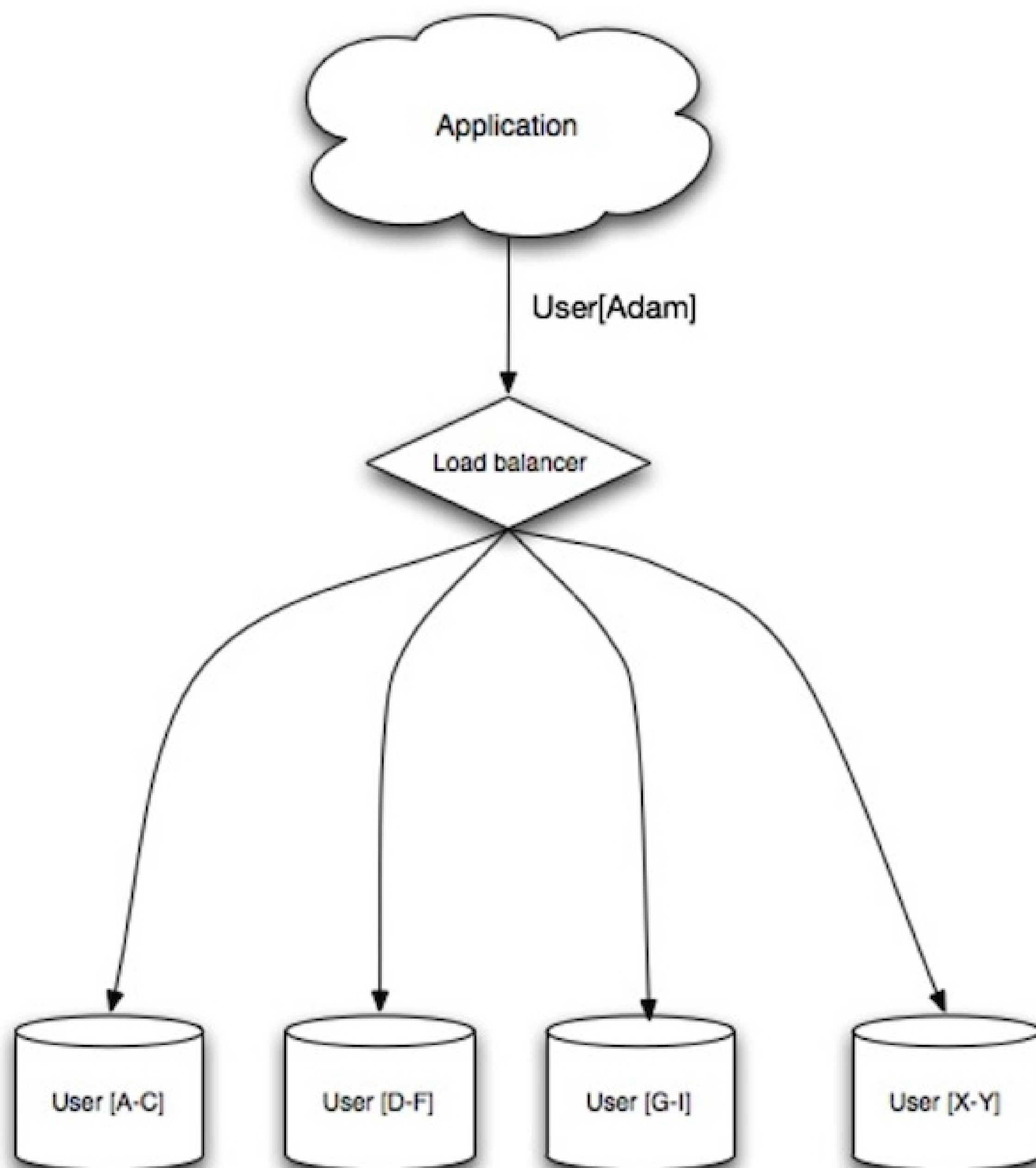
不利之处：联合

- 如果你的数据库模式需要大量的功能和数据表，联合的效率并不好。
- 你需要更新应用程序的逻辑来确定要读取和写入哪个数据库。
- 用 server link 从两个库联结数据更复杂。
- 联合需要更多的硬件和额外的复杂度。

来源及延伸阅读：联合

- 扩展你的用户数到第一个一千万

分片



资料来源：可扩展性、可用性、稳定性、模式

分片将数据分配在不同的数据库上，使得每个数据库仅管理整个数据集的一个子集。以用户数据库为例，随着用户数量的增加，越来越多的分片会被添加到集群中。

类似联合的优点，分片可以减少读取和写入流量，减少复制并提高缓存命中率。也减少了索引，通常意味着查询更快，性能更好。如果一个分片出问题，其他的仍能运行，你可以使用某种形式的冗余来防止数据丢失。类似联合，没有只能串行写入的中心化主库，你可以并行写入，提高负载能力。

常见的做法是用户姓氏的首字母或者用户的地理位置来分隔用户表。

不利之处：分片

- 你需要修改应用程序的逻辑来实现分片，这会带来复杂的 SQL 查询。
- 分片不合理可能导致数据负载不均衡。例如，被频繁访问的用户数据会导致其所在分片的负载相对其他分片高。
    - 再平衡会引入额外的复杂度。基于一致性哈希的分片算法可以减少这种情况。
- 联结多个分片的数据操作更复杂。
- 分片需要更多的硬件和额外的复杂度。

来源及延伸阅读：分片

- 分片时代来临
- 数据库分片架构)
- 一致性哈希

非规范化

非规范化试图以写入性能为代价来换取读取性能。在多个表中冗余数据副本，以避免高成本的联结操作。一些关系型数据库，比如 PostgreSQl 和 Oracle 支持物化视图，可以处理冗余信息存储和保证冗余副本一致。

当数据使用诸如联合和分片等技术被分割，进一步提高了处理跨数据中心的联结操作复杂度。非规范化可以规避这种复杂的联结操作。

在多数系统中，读取操作的频率远高于写入操作，比例可达到 100:1，甚至 1000:1。需要复杂的数据库联结的读取操作成本非常高，在磁盘操作上消耗了大量时间。

不利之处：非规范化

- 数据会冗余。
- 约束可以帮助冗余的信息副本保持同步，但这样会增加数据库设计的复杂度。
- 非规范化的数据库在高写入负载下性能可能比规范化的数据库差。

来源及延伸阅读：非规范化

- 非规范化

SQL 调优

SQL 调优是一个范围很广的话题，有很多相关的书可以作为参考。

利用基准测试和性能分析来模拟和发现系统瓶颈很重要。

- 基准测试 - 用 ab 等工具模拟高负载情况。

- 性能分析 - 通过启用如慢查询日志等工具来辅助追踪性能问题。

基准测试和性能分析可能会指引你到以下优化方案。

改进模式

- 为了实现快速访问，MySQL 在磁盘上用连续的块存储数据。
- 使用 `CHAR` 类型存储固定长度的字段，不要用 `VARCHAR`。
    - `CHAR` 在快速、随机访问时效率很高。如果使用 `VARCHAR`，如果你想读取下一个字符串，不得不先读取到当前字符串的末尾。
- 使用 `TEXT` 类型存储大块的文本，例如博客正文。`TEXT` 还允许布尔搜索。使用 `TEXT` 字段需要在磁盘上存储一个用于定位文本块的指针。
- 使用 `INT` 类型存储高达 2^32 或 40 亿的较大数字。
- 使用 `DECIMAL` 类型存储货币可以避免浮点数表示错误。
- 避免使用 `BLOBS` 存储对象，存储存放对象的位置。
- `VARCHAR(255)` 是以 8 位数字存储的最大字符数，在某些关系型数据库中，最大限度地利用字节。
- 在适用场景中设置 `NOT NULL` 约束来提高搜索性能。

使用正确的索引

- 你正查询（`SELECT`、`GROUP BY`、`ORDER BY`、`JOIN`）的列如果用了索引会更快。
- 索引通常表示为自平衡的 B 树，可以保持数据有序，并允许在对数时间内进行搜索，顺序访问，插入，删除操作。
- 设置索引，会将数据存在内存中，占用了更多内存空间。
- 写入操作会变慢，因为索引需要被更新。
- 加载大量数据时，禁用索引再加载数据，然后重建索引，这样也许会更快。

避免高成本的联结操作

- 有性能需要，可以进行非规范化。

分割数据表

- 将热点数据拆分到单独的数据表中，可以有助于缓存。

调优查询缓存

- 在某些情况下，查询缓存可能会导致性能问题。

来源及延伸阅读

- MySQL 查询优化小贴士
- 为什么 VARCHAR(255) 很常见？
- Null 值是如何影响数据库性能的？

- 慢查询日志

NoSQL

NoSQL 是键-值数据库、文档型数据库、列型数据库或图数据库的统称。数据库是非规范化的，表联结大多在应用程序代码中完成。大多数 NoSQL 无法实现真正符合 ACID 的事务，支持最终一致。

BASE 通常被用于描述 NoSQL 数据库的特性。相比 CAP 理论，BASE 强调可用性超过一致性。

- 基本可用 - 系统保证可用性。
- 软状态 - 即使没有输入，系统状态也可能随着时间变化。
- 最终一致性 - 经过一段时间之后，系统最终会变一致，因为系统在此期间没有收到任何输入。

除了在 SQL 还是 NoSQL 之间做选择，了解哪种类型的 NoSQL 数据库最适合你的用例也是非常有帮助的。我们将在下一节中快速了解下 键-值存储、文档型存储、列型存储和图存储数据库。

键-值存储

> 抽象模型：哈希表

键-值存储通常可以实现 O(1) 时间读写，用内存或 SSD 存储数据。数据存储可以按字典顺序维护键，从而实现键的高效检索。键-值存储可以用于存储元数据。

键-值存储性能很高，通常用于存储简单数据模型或频繁修改的数据，如存放在内存中的缓存。键-值存储提供的操作有限，如果需要更多操作，复杂度将转嫁到应用程序层面。

键-值存储是如文档存储，在某些情况下，甚至是图存储等更复杂的存储系统的基础。

来源及延伸阅读

- 键-值数据库
- 键-值存储的劣势
- Redis 架构
- Memcached 架构

文档类型存储

> 抽象模型：将文档作为值的键-值存储

文档类型存储以文档（XML、JSON、二进制文件等）为中心，文档存储了指定对象的全部信息。文档存储根据文档自身的内部结构提供 API 或查询语句来实现查询。请注意，许多键-值存储数据库有用值存储元数据的特性，这也模糊了这两种存储类型的界限。

基于底层实现，文档可以根据集合、标签、元数据或者文件夹组织。尽管不同文档可以被组织在一起或者分成一组，但相互之间可能具有完全不同的字段。

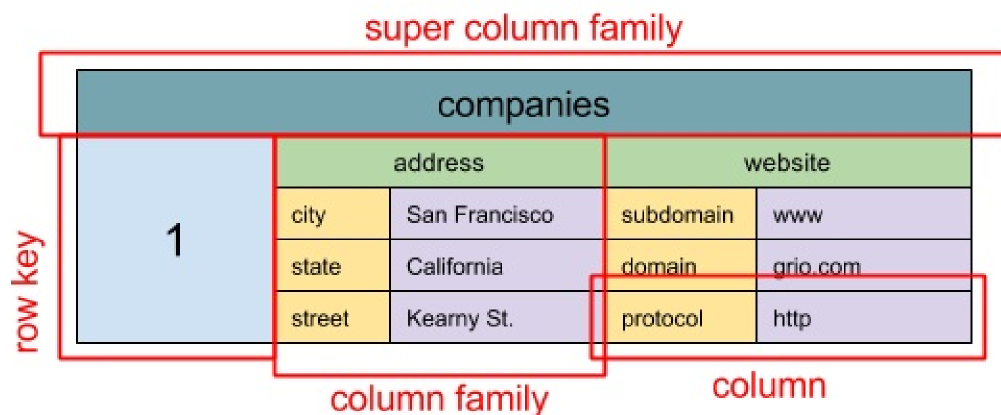
MongoDB 和 CouchDB 等一些文档类型存储还提供了类似 SQL 语言的查询语句来实现复杂查询。DynamoDB 同时支持键-值存储和文档类型存储。

文档类型存储具备高度的灵活性，常用于处理偶尔变化的数据。

来源及延伸阅读：文档类型存储

- 面向文档的数据库
- MongoDB 架构
- CouchDB 架构
- Elasticsearch 架构

列型存储



资料来源: **SQL 和 NoSQL**，一个简短的历史

> 抽象模型：嵌套的 `ColumnFamily<RowKey, Columns<ColKey, Value, Timestamp>>` 映射

类型存储的基本数据单元是列（名／值对）。列可以在列族（类似于 SQL 的数据表）中被分组。超级列族再分组普通列族。你可以使用行键独立访问每一列，具有相同行键值的列组成一行。每个值都包含版本的时间戳用于解决版本冲突。

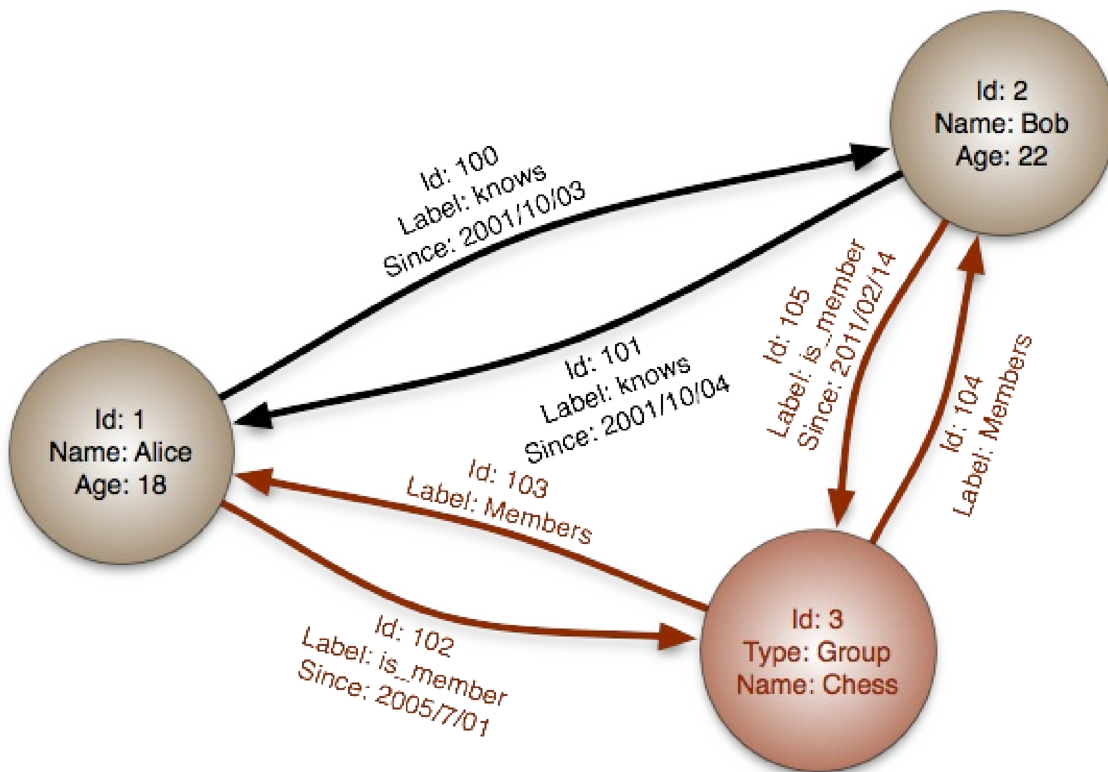
Google 发布了第一个列型存储数据库 Bigtable，它影响了 Hadoop 生态系统中活跃的开源数据库 HBase 和 Facebook 的 Cassandra。像 BigTable，HBase 和 Cassandra 这样的存储系统将键以字母顺序存储，可以高效地读取键列。

列型存储具备高可用性和高可扩展性。通常被用于大数据相关存储。

来源及延伸阅读：列型存储

- SQL 与 NoSQL 简史
- BigTable 架构
- Hbase 架构
- Cassandra 架构

图数据库



资料来源：图数据库

> 抽象模型： 图

在图数据库中，一个节点对应一条记录，一个弧对应两个节点之间的关系。图数据库被优化用于表示外键繁多的复杂关系或多对多关系。

图数据库为存储复杂关系的数据模型，如社交网络，提供了很高的性能。它们相对较新，尚未广泛应用，查找开发工具或者资源相对较难。许多图只能通过 REST API 访问。

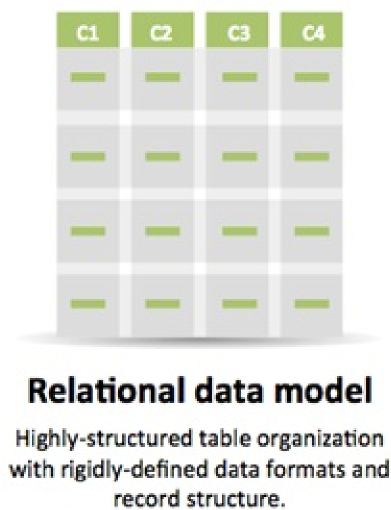
相关资源和延伸阅读：图

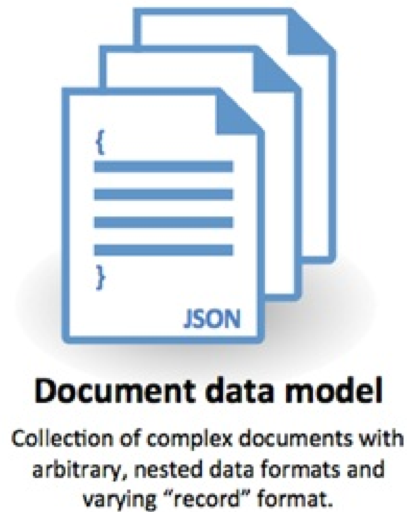
- 图数据库
- Neo4j
- FlockDB

来源及延伸阅读：NoSQL

- 数据库术语解释
- NoSQL 数据库 - 调查及决策指南
- 可扩展性
- NoSQL 介绍
- NoSQL 模式

SQL 还是 NoSQL





资料来源：从 **RDBMS** 转换到 **NoSQL**

选取 **SQL** 的原因:

- 结构化数据
- 严格的模式
- 关系型数据
- 需要复杂的联结操作
- 事务
- 清晰的扩展模式
- 既有资源更丰富：开发者、社区、代码库、工具等
- 通过索引进行查询非常快

选取 **NoSQL** 的原因：

- 半结构化数据
- 动态或灵活的模式
- 非关系型数据
- 不需要复杂的联结操作

- 存储 TB （甚至 PB）级别的数据
- 高数据密集的工作负载
- IOPS 高吞吐量

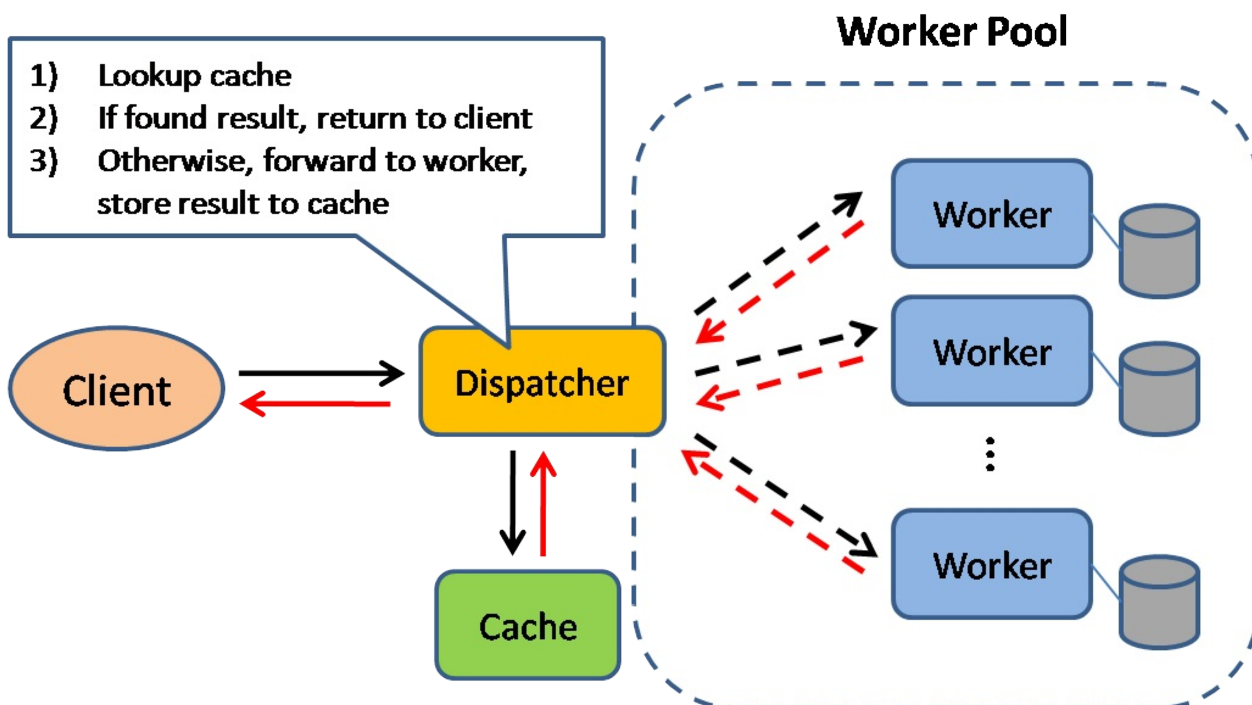
适合 NoSQL 的示例数据：

- 埋点数据和日志数据
- 排行榜或者得分数据
- 临时数据，如购物车
- 频繁访问的（“热”）表
- 元数据／查找表

来源及延伸阅读：**SQL 或 NoSQL**

- 扩展你的用户数到第一个千万
- SQL 和 NoSQL 的不同

缓存



资料来源：可扩展的系统设计模式

缓存可以提高页面加载速度，并可以减少服务器和数据库的负载。在这个模型中，分发器先查看请求之前是否被响应过，如果有则将之前的结果直接返回，来省掉真正的处理。

数据库分片均匀分布的读取是最好的。但是热门数据会让读取分布不均匀，这样就会造成瓶颈，如果在数据库前加个缓存，就会抹平不均匀的负载和突发流量对数据库的影响。

客户端缓存

缓存可以位于客户端（操作系统或者浏览器），服务端或者不同的缓存层。

CDN 缓存

CDN 也被视为一种缓存。

Web 服务器缓存

反向代理和缓存（比如 Varnish）可以直接提供静态和动态内容。Web 服务器同样也可以缓存请求，返回相应结果而不必连接应用服务器。

数据库缓存

数据库的默认配置中通常包含缓存级别，针对一般用例进行了优化。调整配置，在不同情况下使用不同的模式可以进一步提高性能。

应用缓存

基于内存的缓存比如 Memcached 和 Redis 是应用程序和数据存储之间的一种键值存储。由于数据保存在 RAM 中，它比存储在磁盘上的典型数据库要快多了。RAM 比磁盘限制更多，所以例如 least recently used (LRU) 的缓存无效算法可以将「热门数据」放在 RAM 中，而对一些比较「冷门」的数据不做处理。

Redis 有下列附加功能：

- 持久性选项
- 内置数据结构比如有序集合和列表

有多个缓存级别，分为两大类：数据库查询和对象：

- 行级别
- 查询级别
- 完整的可序列化对象
- 完全渲染的 HTML

一般来说，你应该尽量避免基于文件的缓存，因为这使得复制和自动缩放很困难。

数据库查询级别的缓存

当你查询数据库的时候，将查询语句的哈希值与查询结果存储到缓存中。这种方法会遇到以下问题：

- 很难用复杂的查询删除已缓存结果。
- 如果一条数据比如表中某条数据的一项被改变，则需要删除所有可能包含已更改项的缓存结果。

对象级别的缓存

将您的数据视为对象，就像对待你的应用代码一样。让应用程序将数据从数据库中组合到类实例或数据结构中：

- 如果对象的基础数据已经更改了，那么从缓存中删掉这个对象。
- 允许异步处理：workers 通过使用最新的缓存对象来组装对象。

建议缓存的内容：

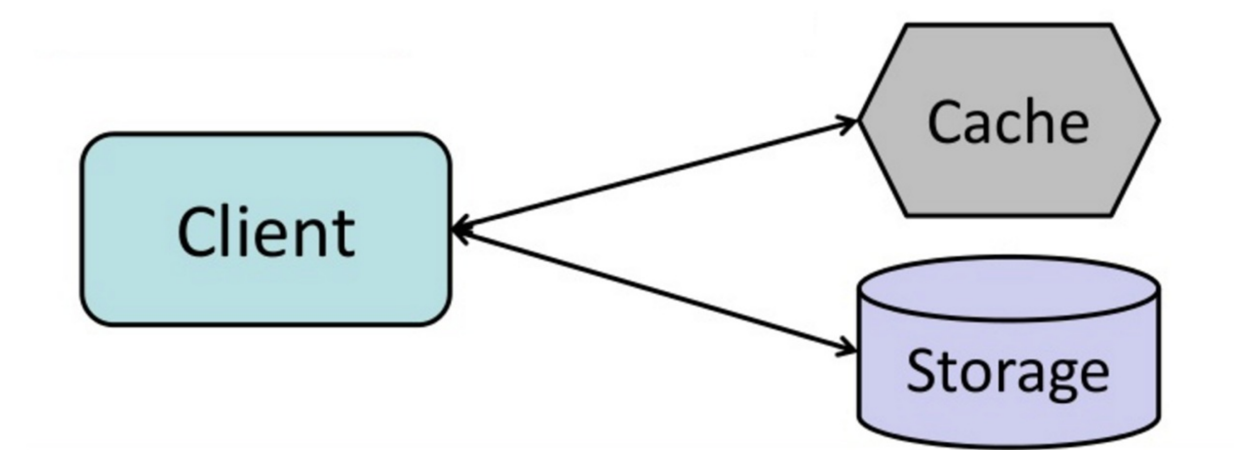
- 用户会话
- 完全渲染的 Web 页面

- 活动流
- 用户图数据

何时更新缓存

由于你只能在缓存中存储有限的数据，所以你需要选择一个适用于你用例的缓存更新策略。

缓存模式



资料来源：从缓存到内存数据网格

应用从存储器读写。缓存不和存储器直接交互，应用执行以下操作：

- 在缓存中查找记录，如果所需数据不在缓存中
- 从数据库中加载所需内容
- 将查找到的结果存储到缓存中
- 返回所需内容

```
def get_user(self, user_id):
    user = cache.get("user.{0}", user_id)
    if user is None:
        user = db.query("SELECT * FROM users WHERE user_id = {0}", user_id)
        if user is not None:
            key = "user.{0}".format(user_id)
            cache.set(key, json.dumps(user))
    return user
```

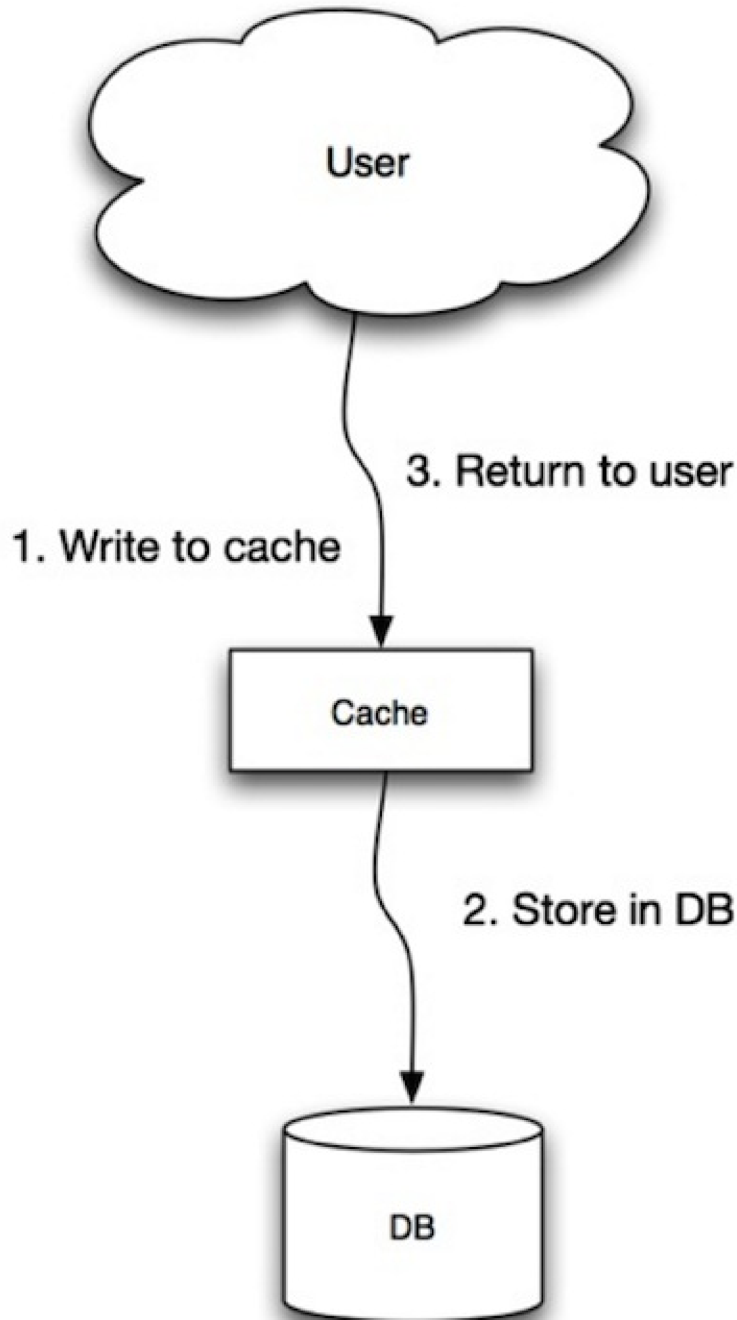
Memcached 通常用这种方式使用。

添加到缓存中的数据读取速度很快。缓存模式也称为延迟加载。只缓存所请求的数据，这避免了没有被请求的数据占满了缓存空间。

缓存的缺点：

- 请求的数据如果不在缓存中就需要经过三个步骤来获取数据，这会导致明显的延迟。
- 如果数据库中的数据更新了会导致缓存中的数据过时。这个问题需要通过设置 TTL 强制更新缓存或者直写模式来缓解这种情况。
- 当一个节点出现故障的时候，它将会被一个新的节点替代，这增加了延迟的时间。

直写模式



资料来源：可扩展性、可用性、稳定性、模式

应用使用缓存作为主要的数据存储，将数据读写到缓存中，而缓存负责从数据库中读写数据。

- 应用向缓存中添加/更新数据
- 缓存同步地写入数据存储
- 返回所需内容

应用代码：

```
set_user(12345, {"foo":"bar"})
```

缓存代码：

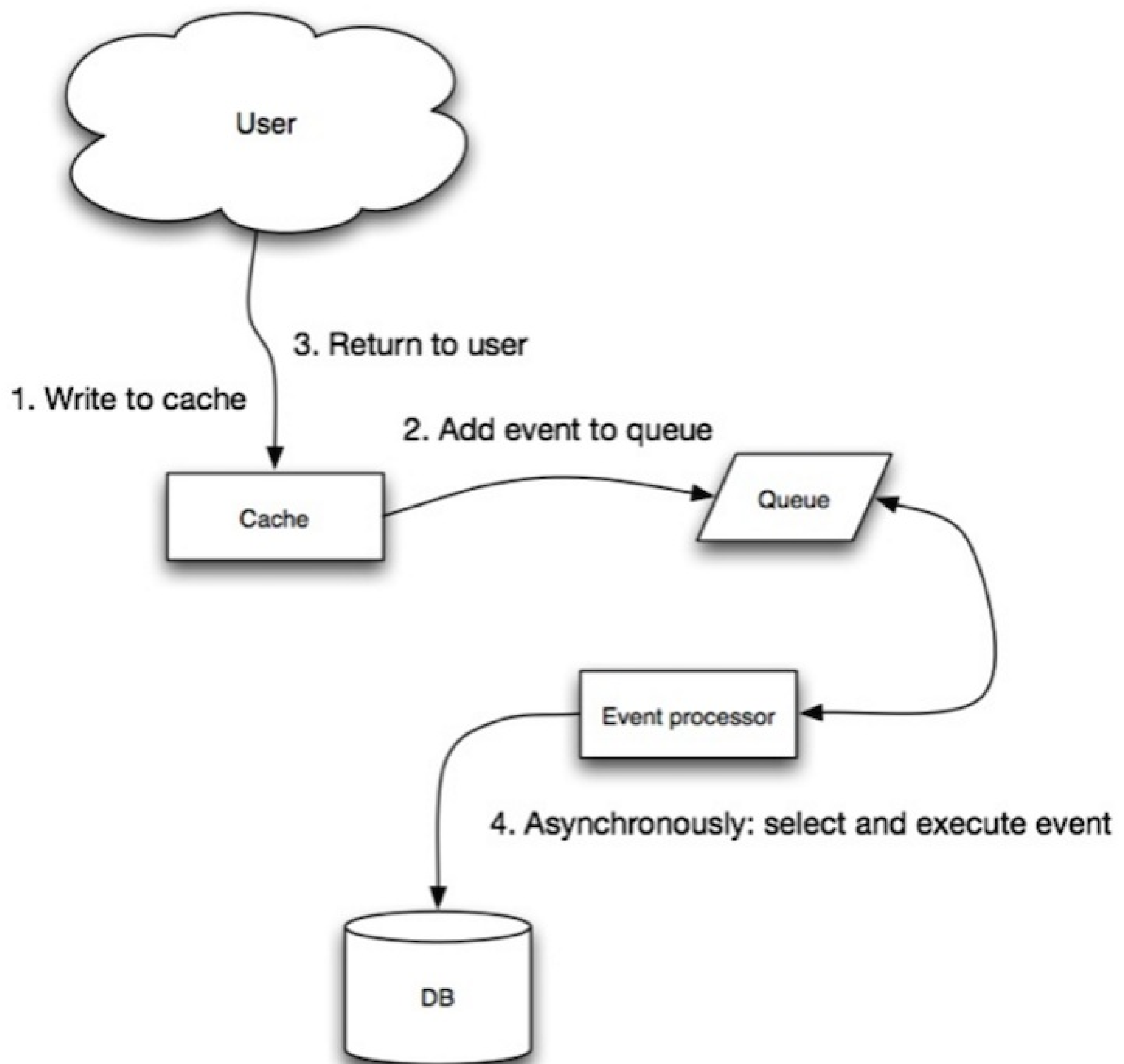
```
def set_user(user_id, values):
    user = db.query("UPDATE Users WHERE id = {0}", user_id, values)
    cache.set(user_id, user)
```

由于存写操作所以直写模式整体是一种很慢的操作，但是读取刚写入的数据很快。相比读取数据，用户通常比较能接受更新数据时速度较慢。缓存中的数据不会过时。

直写模式的缺点：

- 由于故障或者缩放而创建的新的节点，新的节点不会缓存，直到数据库更新为止。缓存应用直写模式可以缓解这个问题。
- 写入的大多数数据可能永远都不会被读取，用 TTL 可以最小化这种情况的出现。

回写模式



资料来源：可扩展性、可用性、稳定性、模式

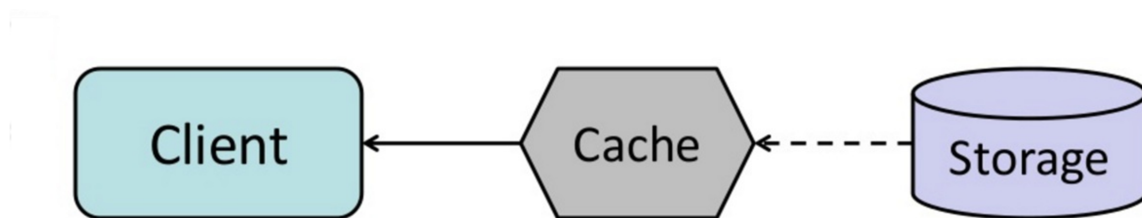
在回写模式中，应用执行以下操作：

- 在缓存中增加或者更新条目
- 异步写入数据，提高写入性能。

回写模式的缺点：

- 缓存可能在其内容成功存储之前丢失数据。
- 执行直写模式比缓存或者回写模式更复杂。

刷新



资料来源：从缓存到内存数据网格

你可以将缓存配置成在到期之前自动刷新最近访问过的内容。

如果缓存可以准确预测将来可能请求哪些数据，那么刷新可能会导致延迟与读取时间的降低。

刷新的缺点：

- 不能准确预测到未来需要用到的数据可能会导致性能不如不使用刷新。

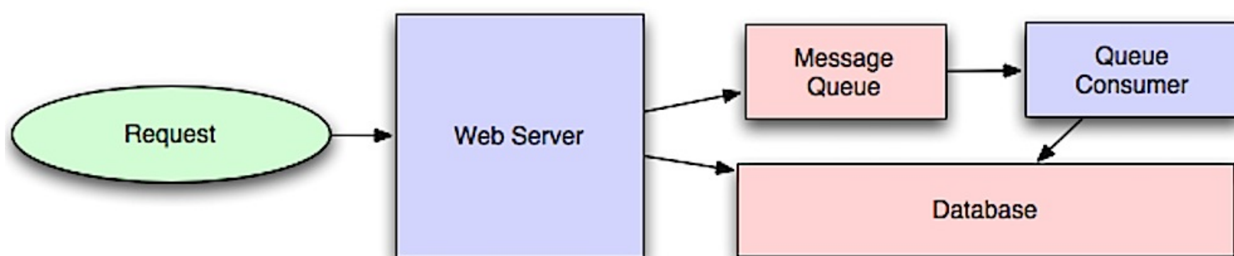
缓存的缺点：

- 需要保持缓存和真实数据源之间的一致性，比如数据库根据缓存无效。
- 需要改变应用程序比如增加 Redis 或者 memcached。
- 无效缓存是个难题，什么时候更新缓存是与之相关的复杂问题。

相关资源和延伸阅读

- 从缓存到内存数据
- 可扩展系统设计模式
- 可缩放系统构架介绍
- 可扩展性，可用性，稳定性和模式
- 可扩展性
- AWS ElastiCache 策略
- 维基百科)

异步



资料来源：可缩放系统构架介绍

异步工作流有助于减少那些原本顺序执行的请求时间。它们可以通过提前进行一些耗时的工作来帮助减少请求时间，比如定期汇总数据。

消息队列

消息队列接收，保留和传递消息。如果按顺序执行操作太慢的话，你可以使用有以下工作流的消息队列：

- 应用程序将作业发布到队列，然后通知用户作业状态
- 一个 worker 从队列中取出该作业，对其进行处理，然后显示该作业完成

不去阻塞用户操作，作业在后台处理。在此期间，客户端可能会进行一些处理使得看上去像是任务已经完成了。例如，如果要发送一条推文，推文可能会马上出现在你的时间线上，但是可能需要一些时间才能将你的推文推送到你的所有关注者那里去。

Redis 是一个令人满意的简单的消息代理，但是消息有可能会丢失。

RabbitMQ 很受欢迎但是要求你适应「AMQP」协议并且管理你自己的节点。

Amazon SQS 是被托管的，但可能具有高延迟，并且消息可能会被传送两次。

任务队列

任务队列接收任务及其相关数据，运行它们，然后传递其结果。 它们可以支持调度，并可用于在后台运行计算密集型作业。

Celery 支持调度，主要是用 Python 开发的。

背压

如果队列开始明显增长，那么队列大小可能会超过内存大小，导致高速缓存未命中，磁盘读取，甚至性能更慢。背压可以通过限制队列大小来帮助我们，从而为队列中的作业保持高吞吐率和良好的响应时间。一旦队列填满，客户端将得到服务器忙活着 HTTP 503 状态码，以便稍后重试。客户端可以在稍后时间重试该请求，也许是指数退避。

异步的缺点：

- 简单的计算和实时工作流等用例可能更适用于同步操作，因为引入队列可能会增加延迟和复杂性。

相关资源和延伸阅读

- 这是一个数字游戏
- 超载时应用背压
- 利特尔法则
- 消息队列与任务队列有什么区别？

通讯

OSI (Open Source Interconnection) 7 Layer Model

Layer	Application/Example	Central Device/ Protocols
Application (7) Serves as the window for users and application processes to access the network services.	**End User layer** Program that opens what was sent or creates what is to be sent Resource sharing • Remote file access • Remote printer access • Directory services • Network management	**User Applications** SMTP
Presentation (6) Formats the data to be presented to the Application layer. It can be viewed as the "Translator" for the network.	**Syntax layer** encrypt & decrypt (if needed) Character code translation • Data conversion • Data compression • Data encryption • **Character Set Translation**	JPEG/ASCII EBDIC/TIFF/GIF PICT
Session (5) Allows session establishment between processes running on different stations.	**Synch & send to ports** (logical ports) Session establishment, maintenance and termination • Session support - perform security, name recognition, logging, etc.	**Logical Ports** RPC/SQL/NFS NetBIOS names
Transport (4) Ensures that messages are delivered error-free, in sequence, and with no losses or duplications.	**TCP** Host to Host, Flow Control Message segmentation • Message acknowledgement • Message traffic control • Session multiplexing PACKET FILTERING	TCP/SPX/UDP
Network (3) Controls the operations of the subnet, deciding which physical path the data takes.	**Packets** ("letter", contains IP address) Routing • Subnet traffic control • Frame fragmentation • Logical-physical address mapping • Subnet usage accounting PACKET FILTERING	**Routers** IP/IPX/ICMP
Data Link (2) Provides error-free transfer of data frames from one node to another over the Physical layer.	**Frames** ("envelopes", contains MAC address) [NIC card — Switch — NIC card] (end to end) Establishes & terminates the logical link between nodes • Frame traffic control • Frame sequencing • Frame acknowledgment • Frame delimiting • Frame error checking • Media access control	**Switch Bridge WAP** PPP/SLIP Land Based Layers
Physical (1) Concerned with the transmission and reception of the unstructured raw bit stream over the physical medium.	**Physical structure** Cables, hubs, etc. Data Encoding • Physical medium attachment • Transmission technique - Baseband or Broadband • Physical medium transmission Bits & Volts	**Hub** Land Based Layers

资料来源：**OSI 7**层模型

超文本传输协议（HTTP）

HTTP 是一种在客户端和服务器之间编码和传输数据的方法。它是一个请求/响应协议：客户端和服务端针对相关内容和完成状态信息的请求和响应。HTTP 是独立的，允许请求和响应流经许多执行负载均衡，缓存，加密和压缩的中间路由器和服务器。

一个基本的 HTTP 请求由一个动词（方法）和一个资源（端点）组成。 以下是常见的 HTTP 动词：

动词	描述	*幂等	安全性	可缓存
GET	读取资源	Yes	Yes	Yes
POST	创建资源或触发处理数据的进程	No	No	Yes，如果回应包含刷新信息
PUT	创建或替换资源	Yes	No	No
PATCH	部分更新资源	No	No	Yes，如果回应包含刷新信息
DELETE	删除资源	Yes	No	No

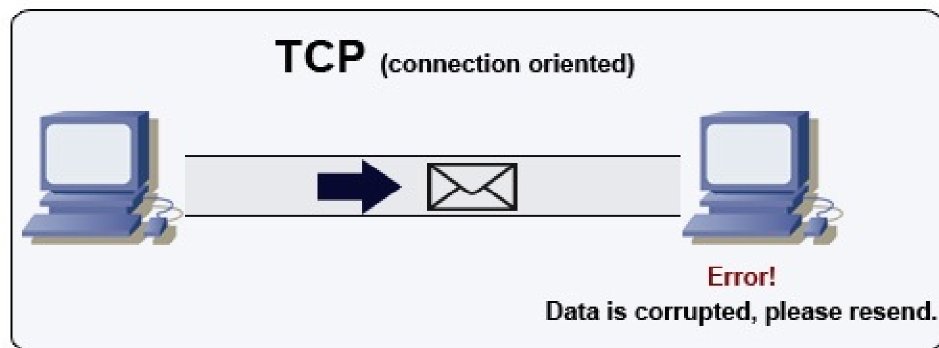
多次执行不会产生不同的结果。

HTTP 是依赖于较低级协议（如 **TCP** 和 **UDP**）的应用层协议。

来源及延伸阅读：HTTP

- README +
- HTTP 是什么？
- HTTP 和 TCP 的区别
- PUT 和 PATCH的区别

传输控制协议（TCP）



资料来源：如何制作多人游戏

TCP 是通过 IP 网络的面向连接的协议。 使用握手建立和断开连接。 发送的所有数据包保证以原始顺序到达目的地，用以下措施保证数据包不被损坏：

- 每个数据包的序列号和校验码。
- 确认包)和自动重传

如果发送者没有收到正确的响应，它将重新发送数据包。如果多次超时，连接就会断开。TCP 实行流量控制)和拥塞控制。这些确保措施会导致延迟，而且通常导致传输效率比 UDP 低。

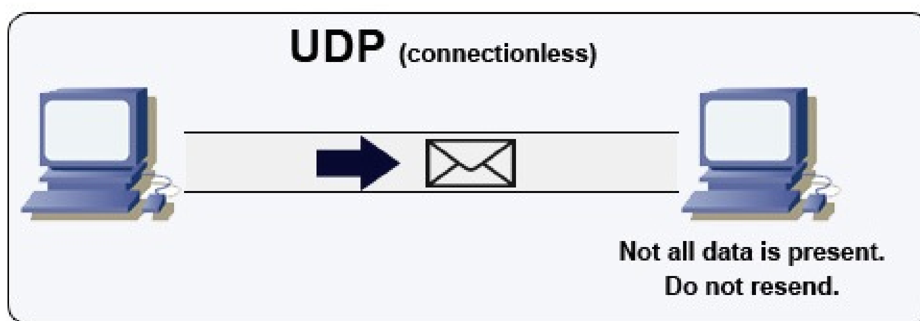
为了确保高吞吐量，Web 服务器可以保持大量的 TCP 连接，从而导致高内存使用。在 Web 服务器线程间拥有大量开放连接可能开销巨大，消耗资源过多，也就是说，一个 memcached 服务器。连接池 可以帮助除了在适用的情况下切换到 UDP。

TCP 对于需要高可靠性但时间紧迫的应用程序很有用。比如包括 Web 服务器，数据库信息，SMTP，FTP 和 SSH。

以下情况使用 TCP 代替 UDP：

- 你需要数据完好无损。
- 你想对网络吞吐量自动进行最佳评估。

用户数据报协议（UDP）



资料来源：如何制作多人游戏

UDP 是无连接的。数据报（类似于数据包）只在数据报级别有保证。数据报可能会无序的到达目的地，也有可能会遗失。UDP 不支持拥塞控制。虽然不如 TCP 那样有保证，但 UDP 通常效率更高。

UDP 可以通过广播将数据报发送至子网内的所有设备。这对 DHCP 很有用，因为子网内的设备还没有分配 IP 地址，而 IP 对于 TCP 是必须的。

UDP 可靠性更低但适合用在网络电话、视频聊天，流媒体和实时多人游戏上。

以下情况使用 UDP 代替 TCP：

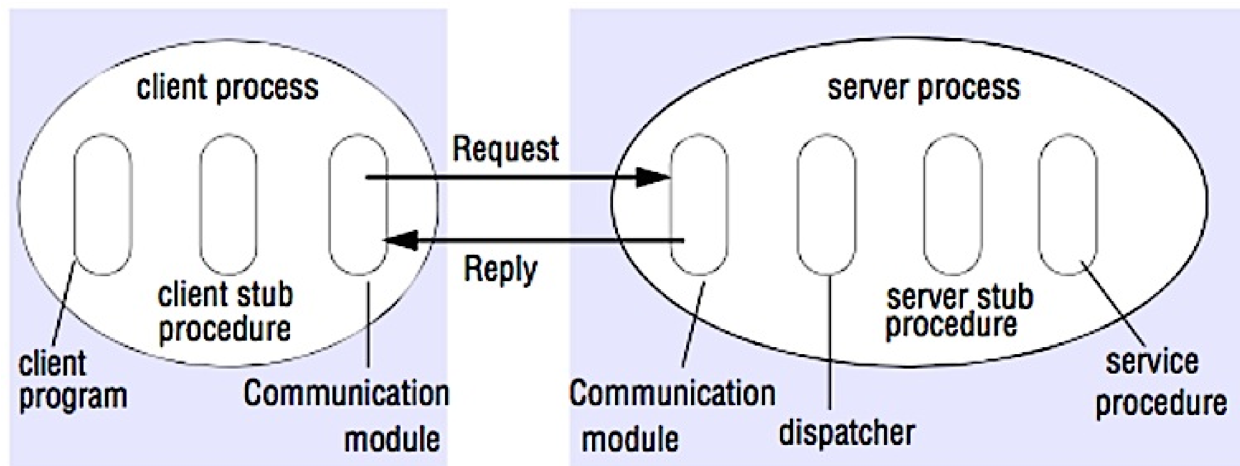
- 你需要低延迟
- 相对于数据丢失更糟的是数据延迟
- 你想实现自己的错误校正方法

来源及延伸阅读：TCP 与 UDP

- 游戏编程的网络
- TCP 与 UDP 的关键区别
- TCP 与 UDP 的不同
- 传输控制协议

- 用户数据报协议
- Memcache 在 Facebook 的扩展

远程过程调用协议（RPC）



Source: Crack the system design interview

在 RPC 中，客户端会去调用另一个地址空间（通常是一个远程服务器）里的方法。调用代码看起来就像是调用的是一个本地方法，客户端和服务器交互的具体过程被抽象。远程调用相对于本地调用一般较慢而且可靠性更差，因此区分两者是有帮助的。热门的 RPC 框架包括 Protobuf、Thrift 和 Avro。

RPC 是一个“请求-响应”协议：

- 客户端程序 ── 调用客户端存根程序。就像调用本地方法一样，参数会被压入栈中。
- 客户端 **stub** 程序 ── 将请求过程的 id 和参数打包进请求信息中。
- 客户端通信模块 ── 将信息从客户端发送至服务端。
- 服务端通信模块 ── 将接受的包传给服务端存根程序。
- 服务端 **stub** 程序 ── 将结果解包，依据过程 id 调用服务端方法并将参数传递过去。

RPC 调用示例：

```
GET /someoperation?data=anId

POST /anotheroperation
{
  "data":"anId";
  "anotherdata": "another value"
}
```

RPC 专注于暴露方法。RPC 通常用于处理内部通讯的性能问题，这样你可以手动处理本地调用以更好的适应你的情况。

当以下情况时选择本地库（也就是 SDK）：

- 你知道你的目标平台。
- 你想控制如何访问你的“逻辑”。
- 你想对发生在你的库中的错误进行控制。
- 性能和终端用户体验是你最关心的事。

遵循 **REST** 的 HTTP API 往往更适用于公共 API。

缺点：RPC

- RPC 客户端与服务实现捆绑地很紧密。
- 一个新的 API 必须在每一个操作或者用例中定义。
- RPC 很难调试。
- 你可能没办法很方便的去修改现有的技术。举个例子，如果你希望在 Squid 这样的缓存服务器上确保 RPC 被正确缓存的话可能需要一些额外的努力了。

表述性状态转移（REST）

REST 是一种强制的客户端/服务端架构设计模型，客户端基于服务端管理的一系列资源操作。服务端提供修改或获取资源的接口。所有的通信必须是无状态和可缓存的。

RESTful 接口有四条规则：

- 标志资源（**HTTP** 里的 **URI**）—— 无论什么操作都使用同一个 URI。
- 表示的改变（**HTTP** 的动作）—— 使用动作, headers 和 body。
- 可自我描述的错误信息（**HTTP** 中的 **status code**）—— 使用状态码，不要重新造轮子。
- **HATEOAS**（**HTTP** 中的**HTML** 接口）—— 你的 web 服务器应该能够通过浏览器访问。

REST 请求的例子：

```
GET /someresources/anId

PUT /someresources/anId
{"anotherdata": "another value"}
```

REST 关注于暴露数据。它减少了客户端／服务端的耦合程度，经常用于公共 HTTP API 接口设计。REST 使用更通常与规范化的方法来通过 URI 暴露资源，通过 header 来表述并通过 GET、POST、PUT、DELETE 和 PATCH 这些动作来进行操作。因为无状态的特性，REST 易于横向扩展和隔离。

缺点：REST

- 由于 REST 将重点放在暴露数据，所以当资源不是自然组织的或者结构复杂的时候它可能无法很好的适应。举个例子，返回过去一小时中与特定事件集匹配的更新记录这种操

作就很难表示为路径。使用 REST，可能会使用 URI 路径，查询参数和可能的请求体来实现。

- REST 一般依赖几个动作（GET、POST、PUT、DELETE 和 PATCH），但有时候仅仅这些没法满足你的需要。举个例子，将过期的文档移动到归档文件夹里去，这样的操作可能没法简单的用上面这几个 verbs 表达。
- 为了渲染单个页面，获取被嵌套在层级结构中的复杂资源需要客户端，服务器之间多次往返通信。例如，获取博客内容及其关联评论。对于使用不确定网络环境的移动应用来说，这些多次往返通信是非常麻烦的。
- 随着时间的推移，更多的字段可能会被添加到 API 响应中，较旧的客户端将会接收到所有新的数据字段，即使是那些它们不需要的字段，结果它会增加负载大小并引起更大的延迟。

RPC 与 REST 比较

操作	RPC	REST
注册	**POST** /signup	**POST** /persons

注销	**POST** /resign
	{
	"personid": "1234"
	}
		DELETE
		/persons/1234

读取用户信息	**GET** /readPerson?personid=1234	**GET** /persons/1234
读取用户物品列表	**GET** /readUsersItemsList? personid=1234	**GET** /persons/1234/items

向用户物品列表添加一项	**POST** /addItemToUsersItemsList
	{
	"personid": "1234";
	"itemid": "456"
	}
		POST
		/persons/1234/items
		{
		"itemid": "456"
		}

更新一个物品	**POST** /modifyItem
	{
	"itemid": "456";
	"key": "value"
	}
		PUT /items/456
		{
		"key": "value"
		}

删除一个物品	**POST** /removeItem
	{
	"itemid": "456"
	}
		DELETE /items/456

资料来源：你真的知道你为什么更喜欢 REST 而不是 RPC 吗

来源及延伸阅读：REST 与 RPC

- 你眞的知道你为什么更喜欢 REST 而不是 RPC 吗
- 什么时候 RPC 比 REST 更合适？
- REST vs JSON-RPC
- 揭开 RPC 和 REST 的神秘面纱
- 使用 REST 的缺点是什么
- 破解系统设计面试
- Thrift
- 为什么在内部使用 REST 而不是 RPC

安全

这一部分需要更多内容。一起来吧！

安全是一个宽泛的话题。除非你有相当的经验、安全方面背景或者正在申请的职位要求安全知识，你不需要了解安全基础知识以外的内容：

- 在运输和等待过程中加密
- 对所有的用户输入和从用户那里发来的参数进行处理以防止 XSS 和 SQL 注入。
- 使用参数化的查询来防止 SQL 注入。
- 使用最小权限原则。

来源及延伸阅读

- 为开发者准备的安全引导
- OWASP top ten

附录

一些时候你会被要求做出保守估计。比如，你可能需要估计从磁盘中生成 100 张图片的缩略图需要的时间或者一个数据结构需要多少的内存。**2** 的次方表和每个开发者都需要知道的一些时间数据（译注：OSChina 上有这篇文章的译文）都是一些很方便的参考资料。

2 的次方表

```
Power           Exact Value         Approx Value        Bytes
---------------------------------------------------------------
7                             128
8                             256
10                           1024   1 thousand           1 KB
16                         65,536                       64 KB
20                      1,048,576   1 million            1 MB
30                  1,073,741,824   1 billion            1 GB
32                  4,294,967,296                        4 GB
40              1,099,511,627,776   1 trillion           1 TB
```

来源及延伸阅读

- 2 的次方

每个程序员都应该知道的延迟数

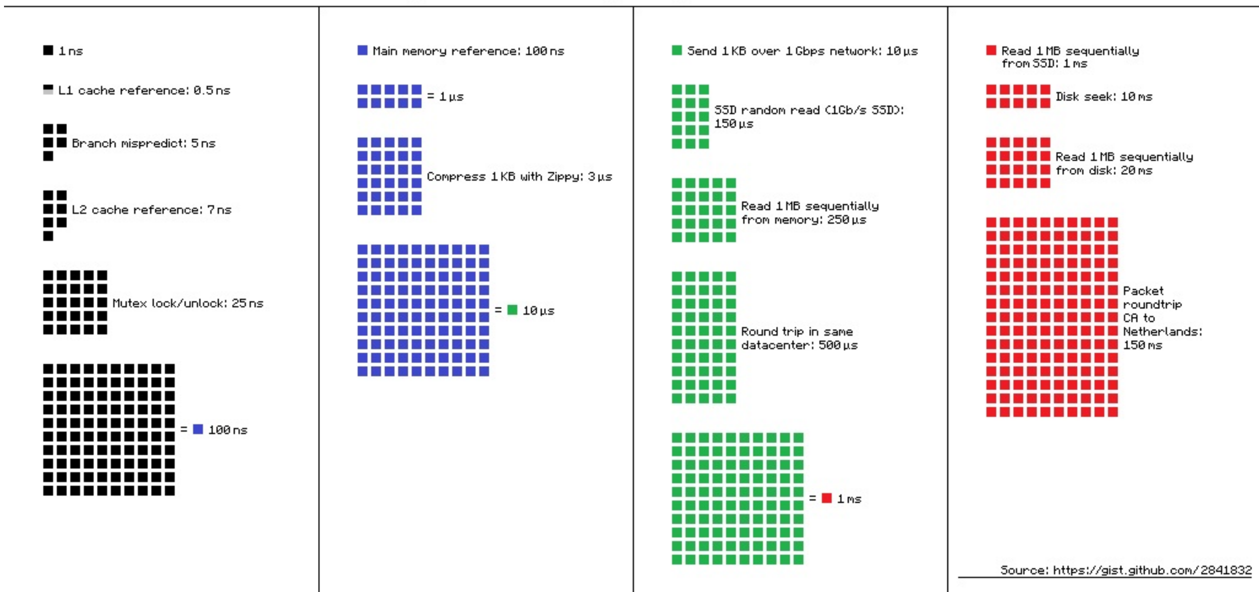
```
Latency Comparison Numbers
--------------------------
L1 cache reference                           0.5 ns
Branch mispredict                            5   ns
L2 cache reference                           7   ns                      14x L1 cache
Mutex lock/unlock                          100   ns
Main memory reference                      100   ns                      20x L2 cache,
 200x L1 cache
Compress 1K bytes with Zippy            10,000   ns       10 us
Send 1 KB bytes over 1 Gbps network     10,000   ns       10 us
Read 4 KB randomly from SSD*           150,000   ns      150 us          ~1GB/sec SSD
Read 1 MB sequentially from memory     250,000   ns      250 us
Round trip within same datacenter      500,000   ns      500 us
Read 1 MB sequentially from SSD*     1,000,000   ns    1,000 us    1 ms  ~1GB/sec SSD,
 4X memory
Disk seek                           10,000,000   ns   10,000 us   10 ms  20x datacente
r roundtrip
Read 1 MB sequentially from 1 Gbps  10,000,000   ns   10,000 us   10 ms  40x memory, 1
0X SSD
Read 1 MB sequentially from disk    30,000,000   ns   30,000 us   30 ms 120x memory, 3
0X SSD
Send packet CA->Netherlands->CA    150,000,000   ns  150,000 us  150 ms

Notes
-----
1 ns = 10^-9 seconds
1 us = 10^-6 seconds = 1,000 ns
1 ms = 10^-3 seconds = 1,000 us = 1,000,000 ns
```

基于上述数字的指标：

- 从磁盘以 30 MB/s 的速度顺序读取
- 以 100 MB/s 从 1 Gbps 的以太网顺序读取
- 从 SSD 以 1 GB/s 的速度读取
- 以 4 GB/s 的速度从主存读取
- 每秒能绕地球 6-7 圈
- 数据中心内每秒有 2,000 次往返

延迟数可视化



来源及延伸阅读

- 每个程序员都应该知道的延迟数 — 1
- 每个程序员都应该知道的延迟数 — 2
- 关于建设大型分布式系统的的设计方案、课程和建议
- 关于建设大型可拓展分布式系统的软件工程咨询

其它的系统设计面试题

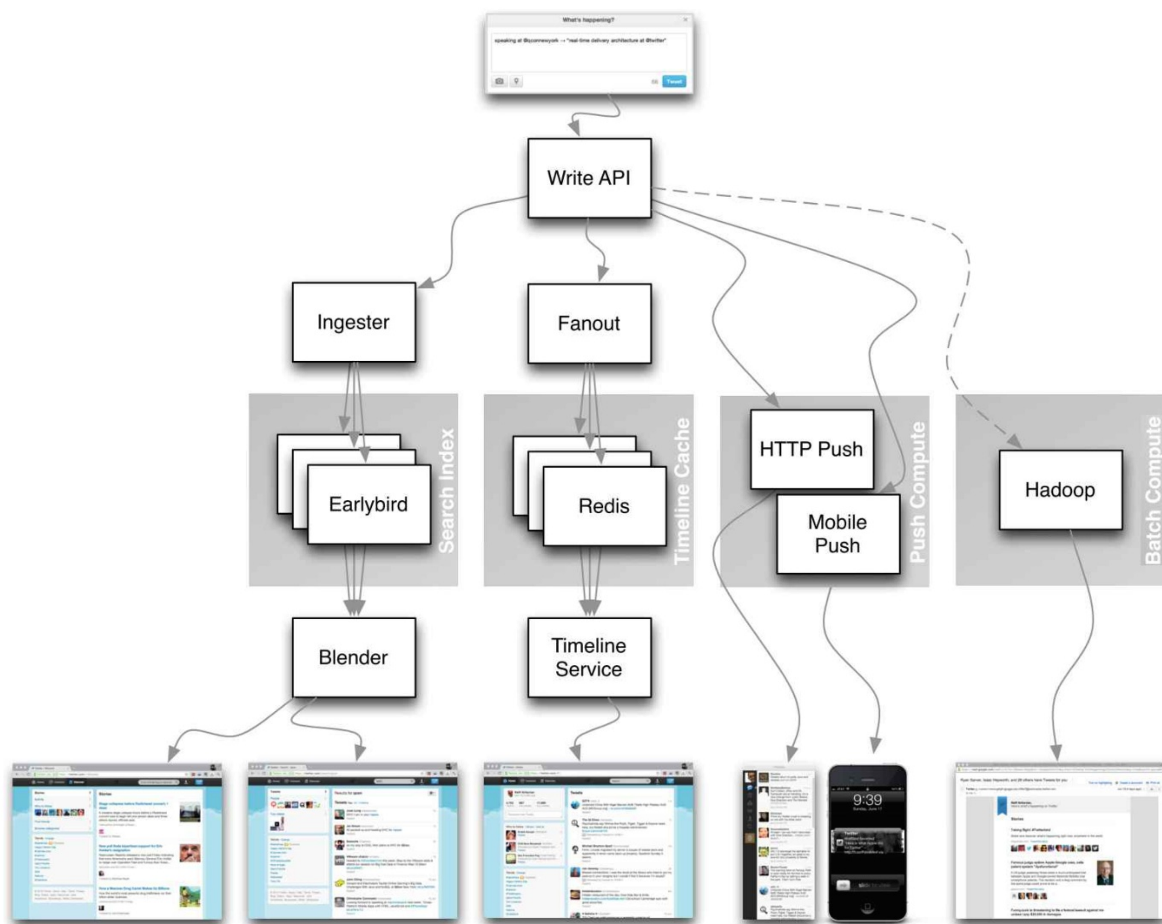
> 常见的系统设计面试问题，给出了如何解决的方案链接

问题	引用
设计类似于 Dropbox 的文件同步服务	youtube.com
设计类似于 Google 的搜索引擎	queue.acm.org stackexchange.com ardendertat.com stanford.edu
设计类似于 Google 的可扩展网络爬虫	quora.com
设计 Google 文档	code.google.com neil.fraser.name
设计类似 Redis 的建值存储	slideshare.net
设计类似 Memcached 的缓存系统	slideshare.net
设计类似亚马逊的推荐系统	hulu.com ijcai13.org
设计类似 Bitly 的短链接系统	n00tc0d3r.blogspot.com

设计类似 WhatsApp 的聊天应用	highscalability.com
设计类似 Instagram 的图片分享系统	highscalability.com highscalability.com
设计 Facebook 的新闻推荐方法	quora.com quora.com slideshare.net
设计 Facebook 的时间线系统	facebook.com highscalability.com
设计 Facebook 的聊天系统	erlang-factory.com facebook.com
设计类似 Facebook 的图表搜索系统	facebook.com facebook.com facebook.com
设计类似 CloudFlare 的内容传递网络	cmu.edu
设计类似 Twitter 的热门话题系统	michael-noll.com snikolov .wordpress.com
设计一个随机 ID 生成系统	blog.twitter.com github.com
返回一定时间段内次数前 k 高的请求	ucsb.edu wpi.edu
设计一个数据源于多个数据中心的服务系统	highscalability.com
设计一个多人网络卡牌游戏	indieflashblog.com buildnewgames.com
设计一个垃圾回收系统	stuffwithstuff.com washington.edu
添加更多的系统设计问题	贡献

真实架构

> 关于现实中真实的系统是怎么设计的文章。



Source: Twitter timelines at scale

不要专注于以下文章的细节，专注于以下方面：

- 发现这些文章中的共同的原则、技术和模式。
- 学习每个组件解决哪些问题，什么情况下使用，什么情况下不适用
- 复习学过的文章

类型	系统	引用
Data processing	**MapReduce** - Google的分布式数据处理	research.google.com
Data processing	**Spark** - Databricks 的分布式数据处理	slideshare.net
Data processing	**Storm** - Twitter 的分布式数据处理	slideshare.net
Data store	**Bigtable** - Google 的列式数据库	harvard.edu
Data store	**HBase** - Bigtable 的开源实现	slideshare.net
Data store	**Cassandra** - Facebook 的列式数据库	slideshare.net
Data store	**DynamoDB** - Amazon 的文档数据库	harvard.edu
Data store	**MongoDB** - 文档数据库	slideshare.net
Data store	**Spanner** - Google 的全球分布数据库	research.google.com
Data store	**Memcached** - 分布式内存缓存系统	slideshare.net
Data store	**Redis** - 能够持久化及具有值类型的分布式内存缓存系统	slideshare.net
File system	**Google File System (GFS)** - 分布式文件系统	research.google.com
File system	**Hadoop File System (HDFS)** - GFS 的开源实现	apache.org
Misc	**Chubby** - Google 的分布式系统的低耦合锁服务	research.google.com
Misc	**Dapper** - 分布式系统跟踪基础设施	research.google.com
Misc	**Kafka** - LinkedIn 的发布订阅消息系统	slideshare.net
Misc	**Zookeeper** - 集中的基础架构和协调服务	slideshare.net
	添加更多	贡献

公司的系统架构

Company	Reference(s)
Amazon	Amazon 的架构
Cinchcast	每天产生 1500 小时的音频
DataSift	每秒实时挖掘 120000 条 tweet
DropBox	我们如何缩放 Dropbox
ESPN	每秒操作 100000 次
Google	Google 的架构
Instagram	1400 万用户，达到兆级别的照片存储 是什么在驱动 Instagram
Justin.tv	Justin.Tv 的直播广播架构
Facebook	Facebook 的可扩展 memcached TAO: Facebook 社交图的分布式数据存储 Facebook 的图片存储
Flickr	Flickr 的架构
Mailbox	在 6 周内从 0 到 100 万用户
Pinterest	从零到每月数十亿的浏览量 1800 万访问用户，10 倍增长，12 名员工
Playfish	月用户量 5000 万并在不断增长
PlentyOfFish	PlentyOfFish 的架构
Salesforce	他们每天如何处理 13 亿笔交易
Stack Overflow	Stack Overflow 的架构
TripAdvisor	40M 访问者，200M 页面浏览量，30TB 数据
Tumblr	每月 150 亿的浏览量
Twitter	Making Twitter 10000 percent faster 每天使用 MySQL 存储2.5亿条 tweet 150M 活跃用户，300K QPS，22 MB/S 的防火墙 可扩展时间表 Twitter 的大小数据 Twitter 的行为：规模超过 1 亿用户
Uber	Uber 如何扩展自己的实时化市场
WhatsApp	Facebook 用 190 亿美元购买 WhatsApp 的架构
YouTube	YouTube 的可扩展性 YouTube 的架构

公司工程博客

> 你即将面试的公司的架构
>
> 你面对的问题可能就来自于同样领域

- Airbnb Engineering
- Atlassian Developers
- Autodesk Engineering
- AWS Blog
- Bitly Engineering Blog
- Box Blogs
- Cloudera Developer Blog
- Dropbox Tech Blog
- Engineering at Quora
- Ebay Tech Blog
- Evernote Tech Blog
- Etsy Code as Craft
- Facebook Engineering
- Flickr Code
- Foursquare Engineering Blog
- GitHub Engineering Blog
- Google Research Blog
- Groupon Engineering Blog
- Heroku Engineering Blog
- Hubspot Engineering Blog
- High Scalability
- Instagram Engineering
- Intel Software Blog
- Jane Street Tech Blog
- LinkedIn Engineering
- Microsoft Engineering
- Microsoft Python Engineering
- Netflix Tech Blog
- Paypal Developer Blog
- Pinterest Engineering Blog
- Quora Engineering
- Reddit Blog
- Salesforce Engineering Blog
- Slack Engineering Blog
- Spotify Labs
- Twilio Engineering Blog
- Twitter Engineering

- Uber Engineering Blog
- Yahoo Engineering Blog
- Yelp Engineering Blog
- Zynga Engineering Blog

来源及延伸阅读

- kilimchoi/engineering-blogs

正在完善中

有兴趣加入添加一些部分或者帮助完善某些部分吗？加入进来吧！

- 使用 MapReduce 进行分布式计算
- 一致性哈希
- 直接存储器访问（DMA）控制器
- 贡献

致谢

整个仓库都提供了证书和源

特别鸣谢：

- Hired in tech
- Cracking the coding interview
- High scalability
- checkcheckzz/system-design-interview
- shashank88/system_design
- mmcgrana/services-engineering
- System design cheat sheet
- A distributed systems reading list
- Cracking the system design interview

联系方式

欢迎联系我讨论本文的不足、问题或者意见。

可以在我的 GitHub 主页上找到我的联系方式

许可